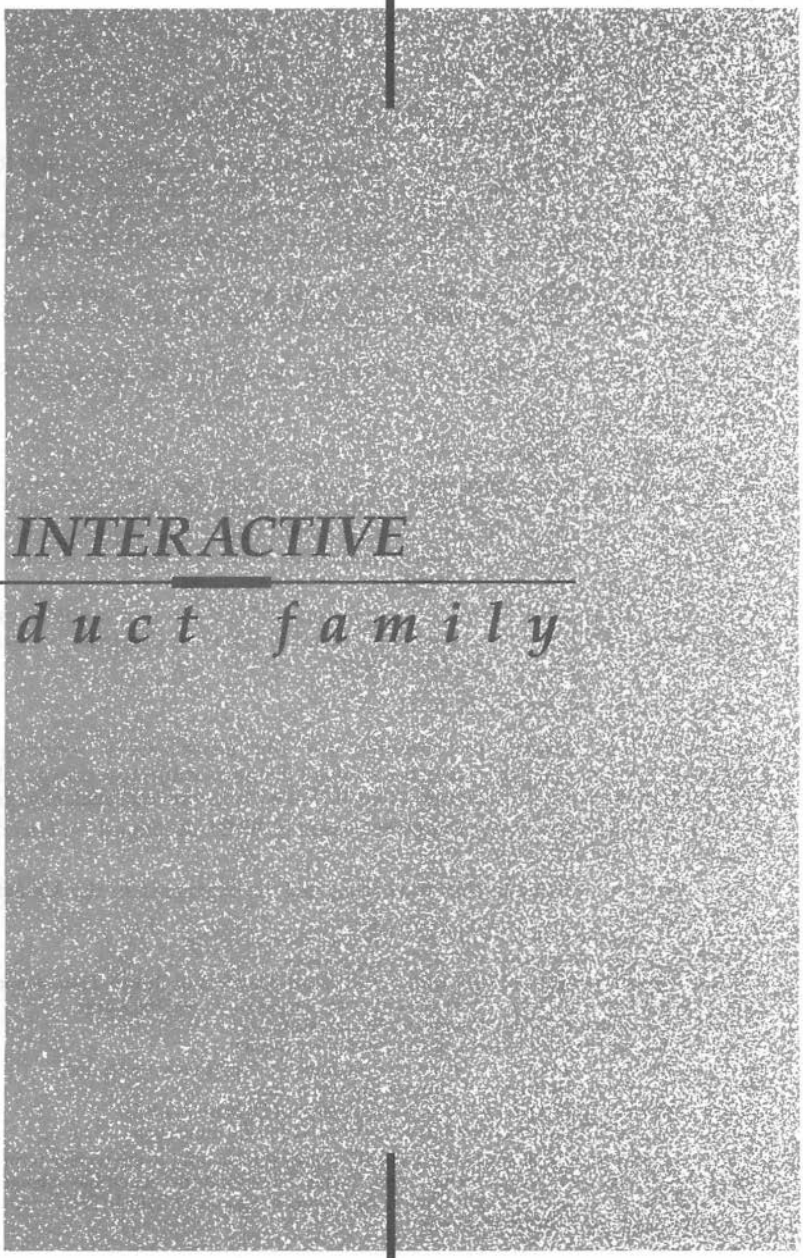
INTERACTIVE NFS* Guide



SunSoft
A Sun Microsystems, Inc. Business

Second printing (February 1992)

SunSoft has reformatted and made editorial changes in the System V NFS Release 3.2.5 Network File System Protocol Specifications and User's Guide, System V NFS Release 3.2.5 User's Reference Manual, System V NFS Release 3.2.5 Programmer's Reference Manual, and the System V NFS Release 3.2.5 Administrator's Reference Manual contained in this guide.

System V NFS* was developed by Lachman Associates, Inc. in cooperation with Sun Microsystems.

SunSoft cannot assume responsibility for any consequences resulting from this publication's use. The information contained herein is subject to change. Revisions to this publication or new editions of it may be issued to incorporate such changes.

INTERACTIVE NFS is derived from System V NFS* developed by Lachman Associates, Inc.

The following trademarks shown as registered are registered in the United States and other countries:

UNIX is a registered trademark of UNIX System Laboratories, Inc.
3COM is a registered trademark of 3COM Corporation.
Yellow Pages is a registered trademark in the United Kingdom of British Telecommunications plc, and may also be a trademark of various telephone companies around the world.
VAX is a trademark of Digital Equipment Corporation.
MS-DOS is a registered trademark of Microsoft Corporation.
NFS and Sun Microsystems are registered trademarks of Sun Microsystems, Inc.
Sun and SunOS are trademarks of Sun Microsystems, Inc.
Ada is a registered trademark of the U.S. Department of Defense A.J.P.O.
Ethernet and XEROX are registered trademarks of XEROX Corporation.

The term "network information service (NIS)" is now used to refer to the service formerly known as Sun Yellow Pages. The functionality remains the same; only the name has changed. The name Yellow Pages is a registered trademark in the United Kingdom of British Telecommunications plc, and may not be used without permission.

INTERACTIVE NFS Guide

CONTENTS

Introduction to the INTERACTIVE NFS* Extension

Welcome to the *INTERACTIVE NFS Guide*. This guide contains the information you need to install, maintain, and use the INTERACTIVE NFS extension. INTERACTIVE NFS is derived from System V NFS* developed by Lachman Associates, Inc. Whether you are an experienced Network File System (NFS) user or brand new to NFS, be sure to read the next few pages of this document. They will tell you what is contained in this guide and how to use the guide to your best advantage.

WHAT'S INCLUDED

The *INTERACTIVE NFS Guide* includes:

- **INTERACTIVE NFS Release Notes**
  Provides a description of the current release of the INTERACTIVE NFS extension.
- **INTERACTIVE NFS User's Manual**
  Provides the basic information and commands needed to share and use files under INTERACTIVE NFS.
- **INTERACTIVE NFS System Administrator's Manual**
  Provides information to set up and maintain INTERACTIVE NFS using the INTERACTIVE TCP/IP transport protocol.
- **INTERACTIVE NFS Protocol Specifications and User's Guide Release 3.2.5**
  Provides an overview of the Network File System and a description of its protocol specifications.
- **INTERACTIVE NFS User's Reference Manual**
  Provides manual entries of interest to INTERACTIVE NFS users.
- **INTERACTIVE NFS Programmer's Reference Manual**
  Provides manual entries of interest to INTERACTIVE NFS programmers.
- **INTERACTIVE NFS Administrator's Reference Manual**
  Provides manual entries of interest to INTERACTIVE NFS system administrators.

- **Reader's Comment Form**
  Provides you with a way to tell us what you like or dislike about this guide and to send us your ideas for making it even better.

WHERE TO BEGIN

The *INTERACTIVE NFS Guide* includes a variety of documents for users at varying levels of experience. Depending on your needs, you may want to use this guide in a number of different ways. The following outline provides some suggested ways to use this guide:

- **If you are a beginner . . .**
  Work through the "INTERACTIVE NFS User's Manual" to learn the commands you need to use the INTERACTIVE NFS extension.

- **If you are an experienced NFS user . . .**
  Read the "INTERACTIVE NFS Protocol Specifications and User's Guide Release 3.2.5," "INTERACTIVE NFS User's Reference Manual," "INTERACTIVE NFS Programmer's Reference Manual," and the "INTERACTIVE NFS Administrator's Reference Manual" for technical details and information about commands.

- **If you are installing the system . . .**
  Read the "INTERACTIVE NFS System Administrator's Manual." Be sure you have read the *INTERACTIVE TCP/IP Guide* and have installed INTERACTIVE TCP/IP on your system.

- **If you want the latest system information . . .**
  Read the "INTERACTIVE NFS Release Notes" included with the INTERACTIVE NFS extension. These notes provide you the latest information on what's new in the most recent release of the INTERACTIVE NFS extension.

- **If you are a programmer or want more detailed information . . .**
  Read the "INTERACTIVE NFS Protocol Specifications and User's Guide Release 3.2.5," "INTERACTIVE NFS User's Reference Manual," "INTERACTIVE NFS Programmer's Reference Manual," and "INTERACTIVE NFS Administrator's Reference Manual."

OVERVIEW OF THE INTERACTIVE NFS EXTENSION

INTERACTIVE NFS is an extension to INTERACTIVE UNIX* System V/386 Release 3.2 that provides a full implementation of Sun Microsystems* Network File System Release 3.5. NFS is a de facto industry standard for transparent file access among differing hardware architectures and operating systems. With the INTERACTIVE NFS extension, users on the network can share files as if they were on the user's local machine, regardless of whether they reside on mainframes, minicomputers, high-performance workstations, or personal computers. Since the protocols are independent of the operating system and transport network, new software and hardware technologies can be integrated easily into the network.

The specification of a "stateless" protocol contributes to the NFS network's reliability and file system integrity. Crash recovery is simple because there is no record-locking state to track or file locking tables to reconstruct. If a particular machine crashes, users can wait for recovery or can use resources elsewhere on the network.

The INTERACTIVE NFS extension provides network services that allow users in heterogeneous computing environments to share files, access remote resources, and mount file systems across a network. Network administration facilities, as well as services for monitoring and maintaining network integrity, are included to make network administration flexible and efficient.

The INTERACTIVE NFS extension interfaces with INTERACTIVE TCP/IP transport layer services. You must have INTERACTIVE TCP/IP installed and operating in order to use the INTERACTIVE NFS extension.

INTERACTIVE NFS FEATURES

The INTERACTIVE NFS extension contains the following features to enhance the INTERACTIVE UNIX Operating System networking environment:

- *Remote Procedure Calls (RPC)*
  Allow remote communication and provide standard message formats for use in specifying higher-level protocols. These provide the facility to specify commands and tasks to be executed on remote machines.

- *eXternal Data Representation (XDR)*
  Provides a vendor-independent representation for data to be exchanged on the network, making heterogeneous communication possible. Byte orders, word length, and floating-point representations appear the same to all machines on the network, circumventing data translation and interpretation problems.
- *Remote EXecution (REX)*
  Allows users and applications to execute commands or programs on remote systems, extending their computing power beyond the capabilities of the local machine.
- *Network Lock Manager*
  Provides file and record locking of remote files, allowing only one user or program at a time to access a particular file or record.
- *Status Monitor*
  Monitors the status of any node on the network to deal with network loading or failure. It works with the Network Lock Manager to provide simple recovery in the event of system or network crashes.
- *Network Information Service (available as an optional subset)*
  The Network Information Service (NIS), formerly called the Yellow Pages*, simplifies network administration by providing centralized database storage of password, group, network, and host information. All users have access to the most current data and can use the NIS subset to look up information about resources by name or type.

DOCUMENTATION REFERENCES

Throughout this guide, the following full documentation titles will be referenced in shortened versions as follows:

Full Title	*Shortened Version*
INTERACTIVE UNIX System V/386 Release 3.2 Operating System Guide	INTERACTIVE UNIX Operating System Guide
INTERACTIVE UNIX System V/386 Release 3.2 Guide for New Users	INTERACTIVE UNIX System Guide for New Users
INTERACTIVE UNIX System V/386 Release 3.2 User's/System Administrator's Reference Manual	INTERACTIVE UNIX System User's/System Administrator's Reference Manual
INTERACTIVE Software Development System Guide and Programmer's Reference Manual	INTERACTIVE SDS Guide and Programmer's Reference Manual

FOR MORE INFORMATION

The INTERACTIVE NFS extension is supported by a complete set of documentation. For a complete listing of all documentation that relates to the INTERACTIVE UNIX Operating System, refer to the "Documentation Roadmap" included in the *INTERACTIVE UNIX Operating System Guide*. For information about installing and maintaining INTERACTIVE TCP/IP, refer to the *INTERACTIVE TCP/IP Guide*.

INTERACTIVE NFS*
Version 2.2
Release Notes
November 1991

1. INTRODUCTION

This document describes the current release of the INTERACTIVE NFS extension for the INTERACTIVE UNIX* Operating System. This release is based on NFS Release 3.5 from Sun Microsystems*. These release notes serve as a caveat sheet, not as an installation guide. Read them before attempting to install the NFS extension on your system. Refer to the "INTERACTIVE NFS System Administrator's Manual" for instructions on installing the INTERACTIVE NFS extension.

2. RELEASE CONTENTS

- Two diskettes containing INTERACTIVE NFS utilities and kernel modules in `sysadm installpkg` format
- *INTERACTIVE NFS Guide*

3. RELEASE REQUIREMENTS

This release of INTERACTIVE NFS works only with the INTERACTIVE UNIX Operating System Version 2.2 or later. For specific subset requirements, refer to "INSTALLING INTERACTIVE NFS" in the "INTERACTIVE NFS System Administrator's Manual" in this guide. The INTERACTIVE Network Information Service (NIS), formerly called the INTERACTIVE Yellow Pages*, is optional and is not included with this package. If you purchased INTERACTIVE NFS, you are entitled to the NIS software. A coupon for NIS is included with each INTERACTIVE NFS software package.

4. NEW FEATURES

The following new features have been added to this release:

- The automounter feature allows NFS file hierarchies to be mounted and unmounted as needed. It uses the daemon program

`automount`(1M) which acts as the NFS server that handles file requests and mounts the appropriate remote hierarchy based on the information specified in the automounter map files. The automounter actually mounts the remote hierarchies under a temporary directory and then uses NFS's symbolic link support to associate the real mount point with the one in the temporary directory. Support for the automounter requires Version 3.0 of the INTERACTIVE UNIX Operating System. See section 8, "The Automounter Guide," in the "INTERACTIVE NFS Protocol Specifications and User's Guide" for more information about this new feature.

- The `exportfs`(1M) command provides finer control over hosts allowed remote access. It is used to export and unexport directories (or local files) to NFS clients. The format of the `/etc/exports` file has changed to support this new feature. See *exports*(4) for details.
- The Lock Manager has been rewritten so that it no longer intercepts file locking on the local system. This greatly enhances its performance and reliability.
- The `rwall`(1M) command is similar to the UNIX System `wall` command except that it is used to broadcast messages to users on remote hosts. `rwall` is supported by the `rwalld`(1M) daemon program which handles the `rwall` requests on the receiving end.

5. BUG FIXES

The following bug fixes have been made in this release:

- Improved Symbolic Link Handling provides better symbolic link handling when used with servers that support symbolic links (BSD, SunOS*).
- The performance of the `lockd` daemon program has been improved. In previous releases, it would core dump and die when the system was heavily loaded. Now that it no longer intercepts all the file locking on the system, the `lockd` program operates more accurately as well.
- Several race conditions in the kernel which would occasionally cause processes (receiving alarm signals while paging) to be killed have been fixed.

6. INTERACTIVE NFS AND SunOS 4.*x*

Since SunOS 4.*x* adds an additional security check to NFS, when connecting a SunOS 4.*x* machine running the INTERACTIVE UNIX Operating System, make sure the setup on Sun* is correct based on Sun's documentation.

7. KNOWN PROBLEMS

7.1 Mounting With 8K Buffers

If you use the `mount` command for INTERACTIVE NFS with no options, the file system will be mounted with 8K read and write size. If you are using a 3COM* 3C501 Ethernet* board, you must mount file systems with a smaller read/write buffer size. For a 3C501 board, type:

```
mount -fNFS,rsize=4096,wsize=4096 <machine>:<dir> <dir>
```

It is not recommended that this board be used with NFS.

7.2 Novell NE-2000 Network Adapters

Due to problems with performance and packet loss under high loads, it is not recommended that the Novell NE-2000 network adapter be used in file servers or other systems with a large amount of NFS traffic.

7.3 Version 3.0 of the INTERACTIVE UNIX System

A minor incompatibility between version 3.0 of the INTERACTIVE UNIX Operating System and version 2.1 of the NIS subset results in slight corruption to the `root` user's `crontab` file after using the `sysadm ypsetup` command. To work around this problem, edit the `/usr/spool/cron/crontabs/root` file. Three lines may begin with the text `L^A^G`. Replace the text with the string `1` on the first line, `11` on the second line, and `21` on the third line.

INTERACTIVE NFS
User's Manual

CONTENTS

INTERACTIVE NFS*
User's Manual

1. INTRODUCTION

INTERACTIVE NFS is an optional extension to the INTERACTIVE UNIX* Operating System that allows several computers to transparently share files across a network. Once the file systems are made available on the network, users on remote machines can access and read or edit the files in the file systems on the remote machines exactly as if the file systems were physically located on their local machine. This means that many different users with login accounts on several different computers can have access to a single set of resource files. Users are not required to keep copies of files that need to be shared on every individual machine. This saves disk space and assures that users are accessing the most current information.

1.1 Before You Begin

This document contains the basic information and commands you may need to use and share files on INTERACTIVE NFS. Before you attempt to use any of the commands or procedures outlined here, you should:

1. Check with your system administrator to verify that the INTERACTIVE NFS extension has been installed and initialized on your system.
2. Read and understand the contents of the "INTERACTIVE UNIX Operating System Primer" in the *INTERACTIVE UNIX System Guide for New Users.*
3. Read this introduction carefully.

If you have problems finding or accessing remote files, see your system administrator.

1.2 Command Syntax

The UNIX System is *case sensitive*, which means that the system always distinguishes between uppercase and lowercase letters. Most

UNIX System commands, options, and arguments are typed in lowercase letters. Options typically begin with a dash (–). Each command, option, or argument consists of one *word*, which is interpreted as a group, or string, of characters surrounded by spaces.

If you make an error when typing a command, use the BACKSPACE key to correct the error. Do *not* use the cursor positioning keys.

Unless otherwise specified, always type the command name first, followed by a space; the desired option or options, each followed by a space; then any arguments, separated by spaces.

Commands in this document will be presented in the following format the first time they appear:

COMMAND NAME	`command name`
FORMAT	`command [ option(s) ]` *argument(s)*
DESCRIPTION	A brief description of what the command does.
OPTIONS	A list of the most useful options and a brief description of each.
ARGUMENTS	Mandatory or optional arguments.

If an argument is not required, it is shown in square brackets [ ]. Options are always "optional," so they are always shown in square brackets [ ]. Only the most common options and arguments are discussed in this document. If there are additional options or arguments available for a particular command that are not presented here, this is indicated by the phrase "Not presented in this document." For a complete listing of the available options and arguments for a command, refer to the reference manuals for your system.

1.3 Basic NFS Terms

NFS allows users on many different computers to share file systems or *resources* across a network. A machine in an NFS environment that is configured to share file systems is called a *host*. File systems are made available to the users on other hosts on the network by the

system administrator. This process is called *exporting*. Once a file system on another computer has been exported, the system administrator of your local machine can make it available to you by *mounting* it on one of your local directories.

Each remote file system available to you has a *mount path name*. This consists of the name of the machine where the file system is located, a colon, and the name of the file system, for example, `scotty:/src`. The file system does not actually leave the machine it resides on, but once it is exported and mounted on your local machine, you can access and read or edit the files in the remote file systems exactly as if the file systems were physically present on your local machine.

2. USER NETWORKING COMMANDS

Sharing file systems across a network using the INTERACTIVE NFS extension is almost as easy as working with files on your own machine. There are only a few special commands you will need to know.

2.1 Listing Local Mounted File Systems (`/etc/mount`)

To find out what file systems are currently available (mounted) on your local machine, use the `/etc/mount` command.

COMMAND NAME	`/etc/mount`
FORMAT	`/etc/mount`
DESCRIPTION	Lists all file systems mounted on the local machine; if local, the device on which it is mounted, or if remote, its file system mount path name; the permissions for each, whether each is a local or a remote resource, and the time each was mounted.
OPTIONS	None.
ARGUMENTS	None.

This command lists all file systems mounted on your local machine. If the file system is a local one (one that is physically located on your machine), it gives the name of the device on which the file system is mounted. If it is a remote file system, `/etc/mount` lists the mount path name instead, since the file system is not physically loaded onto any local device. `/etc/mount` also gives the permissions for a file system, whether it is a remote or a local resource, and the time each was mounted. When you type `/etc/mount`, your screen will look similar to this:

```
$ /etc/mount
/ on /dev/dsk/0s1 read/write on Mon Mar 14 09:15:05 1988
/usr on /dev/dsk/0s3 read/write on Mon Mar 14 09:17:50 1988
/usr2 on /dev/dsk/0s4 read/write on Mon Mar 14 09:17:51 1988
/usr2/test on ollie:/src/test read/remote on Wed Mar 23 11:17:52 1988
/usr2/rel/src on scotty:/src read/write/remote on Mon Mar 21 15:48:44 1988
```

For example, the first listing gives you the following information:

File System Name on Local Machine	*Device Name*	*Permissions and Type of File System*	*Date and Time of Mounting on Local Machine*
/	/dev/dsk/0s1	read/write	Mon Mar 14 09:15:05 1988

The local `root` file system (`/`) is mounted on `/dev/dsk/0s1`. This file system is accessed by typing `cd /`, so you do not need to be concerned with the device it is mounted on. (You can refer to "System Administration for New Users of the INTERACTIVE UNIX Operating System" in the *INTERACTIVE UNIX System Guide for New Users* if you want to learn more about mounting file systems and the naming conventions used for devices.) Since the output of the command does not say `remote` after the permissions (`read/write`), this is a local file system.

The last listing gives this information:

File System Name on Local Machine	*Remote Mount Path Name*	*Permissions and Type of File System*	*Date and Time of Mounting on Local Machine*
/usr/newrel/src	scotty:/src	read/write/remote	Mon Mar 21 15:48:44 1988

The file system named `/src` that is located on the machine named `scotty` is available on your local machine as a file system named `/usr/newrel/src`. If you want to look at the files and directories in `scotty:/src`, type `cd /src/newrel/src` to "go" to that directory and then type `ls` to see a listing of its contents, just as you do to look at local directories.

If a file system you need to use is not currently mounted on your local machine, see your system administrator. You must be the `root` user to mount a file system when running the INTERACTIVE NFS extension on your system.

2.2 Listing Machines With Remotely Mounted File Systems (/etc/showmount)

If you want to list all the machines in the domain that have mounted remote file systems, use the `/etc/showmount` command.

COMMAND NAME	`/etc/showmount`
FORMAT	`/etc/showmount [ -dae ] [`*hostname*` ]`
DESCRIPTION	List all the machines with remotely mounted file systems. If the name of another machine is used as an argument, list the remote machines that have mounted file systems from that machine.
OPTIONS	`-e` Print the list of exported file systems. `-a` Print all remote mounts in the format `hostname:directoryname`. `-d` List directories that have been remotely mounted by other machines.
ARGUMENTS	The name of a particular machine. If no machine name is given, the local machine name is assumed.

Used with no options, the `/etc/showmount` command lists all the machines on your domain that have remotely mounted file systems. For example, if you want to see if the computer called `scotty` is currently on the domain and using any remote file systems, type `/etc/showmount`:

```
$ /etc/showmount
marlon  scotty  ollie  stan
```

There are currently four machines on the domain with remotely mounted file systems; one of them is `scotty`.

2.3 Listing Available File Systems (/etc/showmount -e)

To find out which file systems from your local machine are available to other users on your domain, type `/etc/showmount -e`. Your screen will look similar to this:

```
/src/test
/usr/sales
/usr/offices/newinfo
```

If you want to know whether or not a particular file system is available (has been exported) from another machine on your domain, type `/etc/showmount -e` *machinename.* For example, to determine whether the file system `/usr2/pubs` is available from the machine named `stan`, type:

```
$ /etc/showmount -e stan

  /src/mover
  /test/newrel
  /usr2/pubs
```

Three file systems are available from `stan`, one of which is `/usr2/pubs`.

2.4 Listing All Remotely Mounted File Systems (`/etc/showmount -a`)

To find out all the remotely mounted file systems on the domain, type `/etc/showmount -a`. For example, to determine whether or not the file system called `/usr2/pubs` is currently available to Gerald, who uses the machine named `marlon`, type `/etc/showmount -a`. Your screen will look similar to this:

```
$ /etc/showmount -a

marlon:/src/test
marlon:/src
ollie:/usr/sales
ollie:/usr/offices/newinfo
ollie:/usr2/pubs
scotty:/usr/sales
scotty:/test/newrel
scotty:/usr/offices/newinfo
stan:/usr/offices/newinfo
```

The machine `marlon` does not have the `/usr2/pubs` file system mounted on it at this time.

If you type the name of a particular machine as a second argument, the system lists the machines on the network that have remotely mounted a file system from that machine. For example, to determine what machines on the network have the file system `/src/test` from the machine named `ollie` available, type:

```
$ /etc/showmount -a ollie
```

Your screen will look similar to this:

```
marlon:/src/test
scotty:/usr/bin
bob:/usr/bin
```

`/usr/bin` and `/src/test` are the only two file systems available from `ollie` at this time. Only the machine named `marlon` has `/src/test` available at this time.

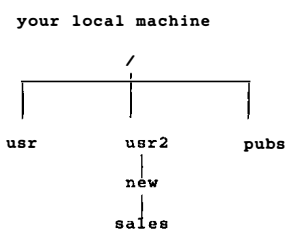
2.5 Using UNIX System Commands on Remote Resources

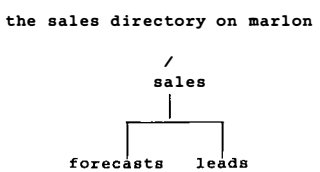
Once you know what directories are available to you, you can use standard UNIX System commands (`cd`, `pwd`, `ls`, and so on) to access a remote directory and manipulate the files and directories located there. For example, to access the remote directory named `/sales` from the machine named `marlon`, which is mounted on your local machine on the directory `/usr2/new/sales`, type:

```
cd /usr2/new/sales
```

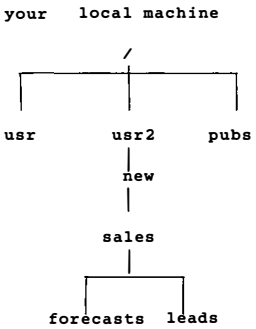
You can now type `ls` to see a list of the remote files and directories available to you. If there is `write` permission on the files as well, you can use local editors to edit the remote files just as you would any other file on your local machine.

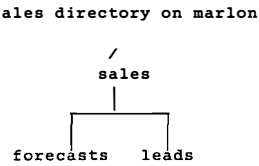
The `/sales` directory is physically located on `marlon` and really has no connection with your local machine. Before the system administrator mounts `/sales` on `/usr2/new/sales` on your local machine, it looks like this:





But, after your system administrator mounts the remote `sales` directory on your local directory `/usr2/new/sales`, you can access the `forecasts` and `leads` files as if they were physically present on your local machine.


your local machine
/
usr
usr2
pubs
new
sales
forecasts
leads


the sales directory on marlon
/
sales
forecasts
leads

GLOSSARY

case sensitive
: Distinguishes between uppercase and lowercase letters.

exporting
: The method by which a file system is made available for sharing with remote machines.

host
: A machine in an NFS environment that is configured to share file systems. The database files are actually edited only on this machine.

mount
: To make a directory available to users on a particular machine.

mount path name
: The full path name of a remote directory that has been exported, consisting of the name of the machine where the file system is located, a colon, and the name of the file system, for example, `stan:/src/newrel`.

resource
: A file system that is shared on the NFS network.

word
: A series of characters surrounded by spaces.

INDEX

INTERACTIVE NFS
System Administrator's Manual

CONTENTS

INTERACTIVE NFS*
System Administrator's Manual

1. INTRODUCTION

INTERACTIVE NFS is an optional extension to the INTERACTIVE UNIX* Operating System that allows users to share file systems transparently across a Transmission Control Protocol/Internet Protocol (TCP/IP) network protocol suite.

TCP/IP is a widely used networking protocol suite; INTERACTIVE TCP/IP can be purchased as an optional extension to the INTERACTIVE UNIX Operating System. INTERACTIVE TCP/IP must be installed on your machine in order to use the INTERACTIVE NFS extension. The applications program libraries provided with INTERACTIVE TCP/IP allow use of the following facilities: TELNET and FTP commands for logging into remote machines and transferring files between computers; several of the Berkeley 4.3 `r`-utilities, including `rsh`, `rlogin`, and `rcp`; and a C-callable network programming library compatible with the Berkeley 4.3 socket system calls for porting existing applications to board-based TCP/IP networking.

The INTERACTIVE NFS extension allows several computers to transparently share file systems across a network. Once the file systems are shared, they can be accessed and read or edited by users on remote machines exactly as if the file systems were physically located on the local machine.

The INTERACTIVE NFS extension also provides an optional subset called the *Network Information Service*. This allows a common distributed database of system files (for example, the `/etc/passwd`, `/etc/group`, and `/etc/hosts` files) to be maintained across all the machines on the network. Refer to the *INTERACTIVE Network Information Service Guide* (formerly called the *INTERACTIVE Yellow Pages Guide*) that accompanies the optional subset software for information about installing, maintaining, and using the Network Information Service facility.

1.1 Purpose of This Document

The instructions in this document provide the information required to set up and maintain the INTERACTIVE NFS extension using the INTERACTIVE TCP/IP package.

Refer to the following documents in the *INTERACTIVE TCP/IP Guide* that accompanies your INTERACTIVE TCP/IP package for more information about installing and using the facilities it provides:

- "Introduction to INTERACTIVE TCP/IP"
- "INTERACTIVE TCP/IP Networking Primer"
- "INTERACTIVE TCP/IP System Administrator's Manual"
- "INTERACTIVE TCP/IP Programmer's Supplement"

The following documentation is also available to support your system.

- *INTERACTIVE UNIX System User's/System Administrator's Reference Manual*
- *INTERACTIVE UNIX System V/386 Release 3.2 User's Guide*
- *INTERACTIVE SDS Guide and Programmer's Reference Manual*
- *UNIX System V/386 Release 3.2 Programmer's Guide*

These documents may be ordered separately from INTERACTIVE Systems Corporation or your sales representative.

2. INSTALLATION REQUIREMENTS

Before you install the INTERACTIVE NFS extension, the following requirements must be met:

- The Kernel Configuration subset must be installed on your machine. (See the "INTERACTIVE UNIX Operating System Installation Instructions" in the *INTERACTIVE UNIX Operating System Guide* for information on how to do this.)
- The INTERACTIVE TCP/IP optional subset must be installed on your machine. (Refer to the "INTERACTIVE TCP/IP System Administrator's Manual" in the *INTERACTIVE TCP/IP Guide* for information on how to do this.)
- The files installed and maintained using the `sysadm` TCP/IP Management Menu (`/etc/hosts.equiv` and `/etc/hosts`) must be consistent across all machines on the network. (For information about maintaining these files, refer to section 3 of the "INTERACTIVE TCP/IP System Administrator's Manual" in the *INTERACTIVE TCP/IP Guide*.
- 4 MB of RAM must be available.
- If your disk is partitioned according to the system defaults, the following disk space is required:
  - 750 blocks on the `root` file system.
  - 2500 blocks on the `/usr` file system.

3. HARDWARE REQUIREMENTS

There are no hardware requirements specifically for INTERACTIVE NFS. The only hardware requirements that apply are those for INTERACTIVE TCP/IP.

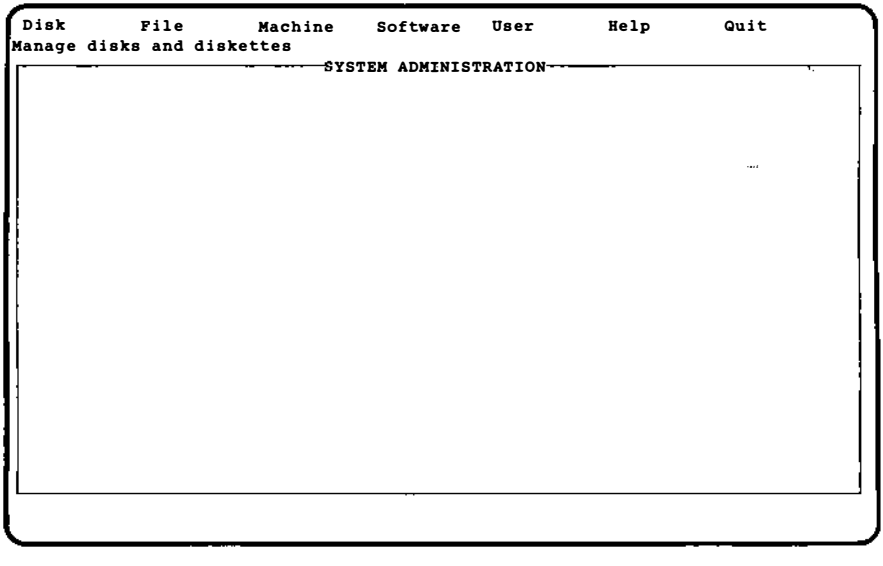
4. INSTALLING INTERACTIVE NFS

The INTERACTIVE NFS extension is delivered on two diskettes. The optional Network Information Service subset is available separately; it does not have to be installed in order to install and use INTERACTIVE NFS.

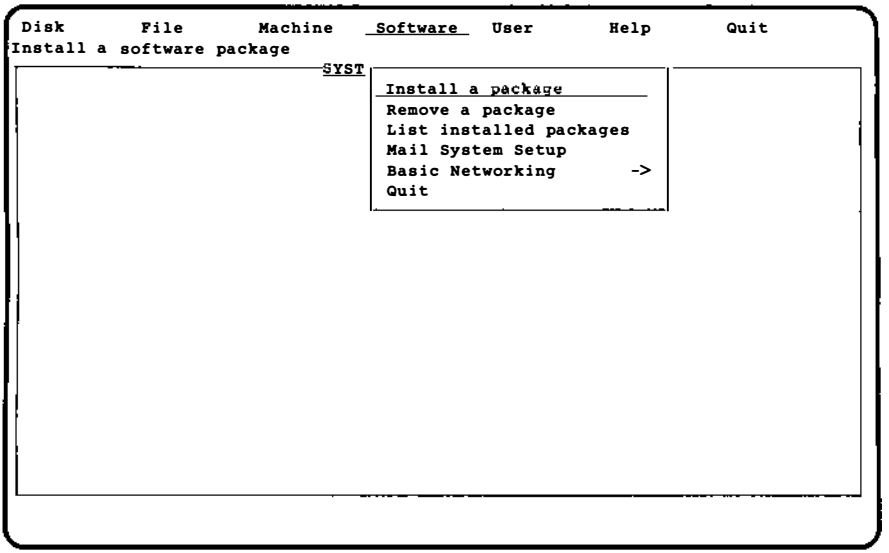
The INTERACTIVE NFS extension is installed using the system administration menus. This subset must be installed on all machines in the planned network.

☛ If NFS is already running, type `init 2` before installing the new version of NFS.

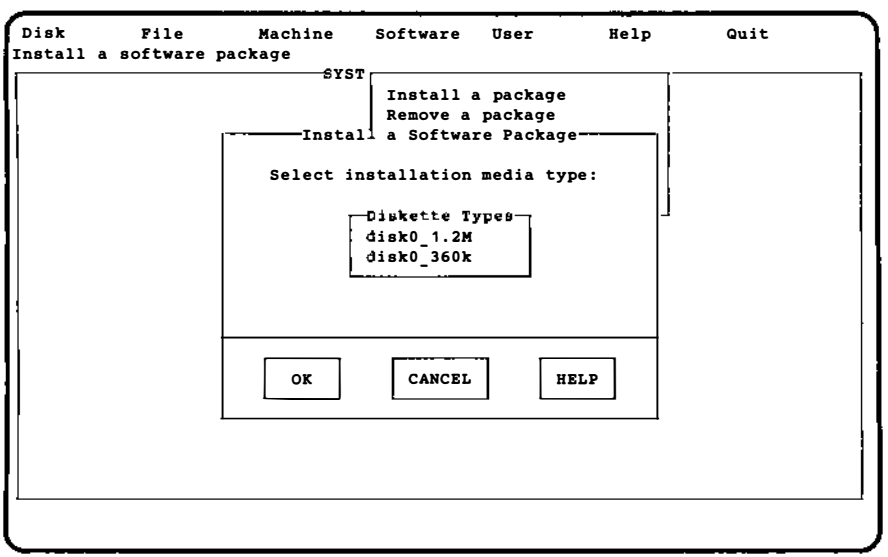
1. Use the System Administration command, `sysadm`, or log in as `sysadm` to access the bar menu. Your screen will look similar to this:



2. Use the left and right arrow keys to move to `Software`, and press ENTER to access the `Software` menu. Your screen will then look similar to this:

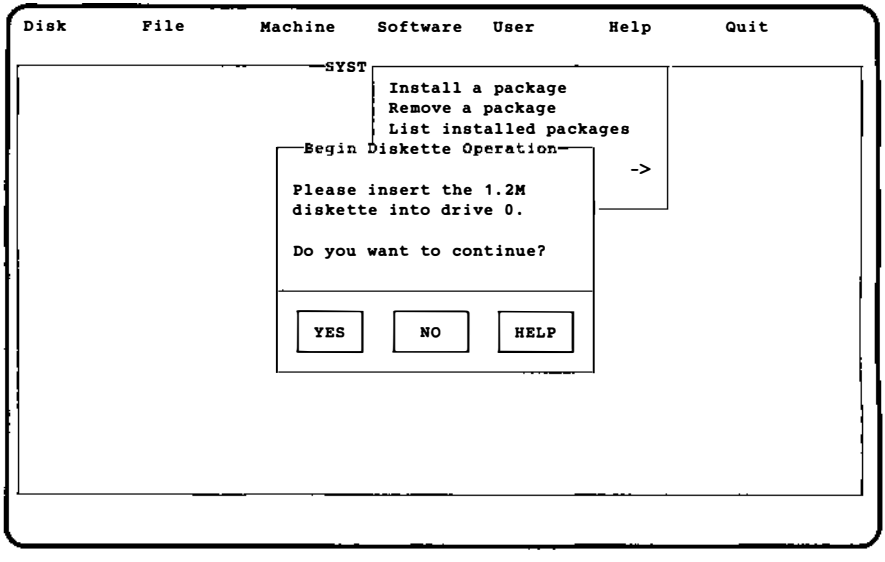


3. Press **ENTER** to select `Install a package`. Your screen will look similar to this:

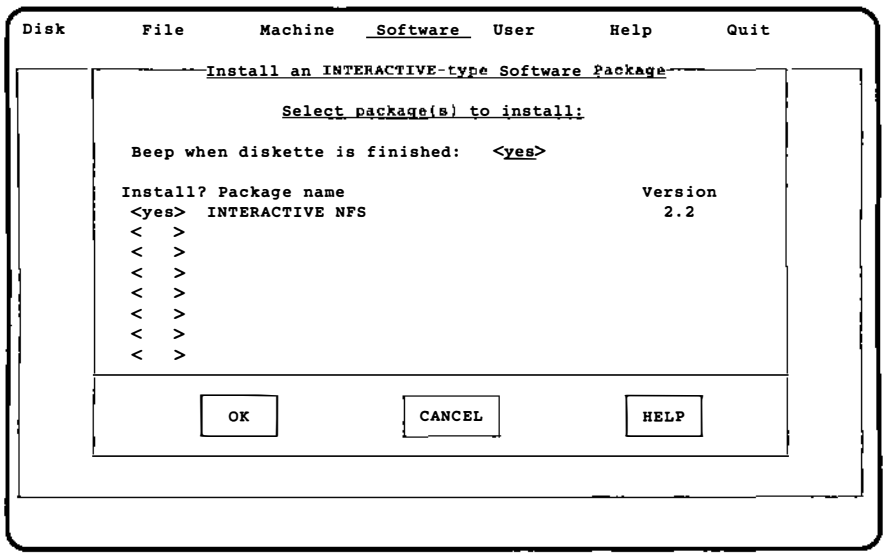


4. Use the up and down arrow keys to move to the type of diskette you are using, and press **ENTER** to select it. Press

ENTER again while on the OK button to move to the next screen. The system will then ask:



5. Insert the first *INTERACTIVE NFS Extension* diskette into the diskette drive and press ENTER while the YES button is highlighted to continue the installation. Your screen will look similar to this:



Press **ENTER** twice to start the installation. Your screen will look similar to this:

```
Running dependencies routine ....
Running copyright routine ....
Copyright (c) 1988-1991 Interactive Systems Corp.
All Rights Reserved.
INTERACTIVE NFS is derived from System V NFS(TM) developed
by Lachman Associates, Inc.
NFS is a trademark of Sun Microsystems, Inc.
Installing INTERACTIVE NFS - Version 2.2
The following files are being installed:
/
/usr
/usr/lib
/usr/lib/librpcsvc.a
/usr/lib/librpc.a
      .
      .
      .
/etc/biod
1916 blocks
File uncompression in progress ....

Please remove the floppy from the drive and insert
The INTERACTIVE NFS - Version 2.2 Disk Number 2.
Type <return> when ready:
```

6. Remove the first diskette, insert the second one, and press **ENTER** to continue. When the installation has finished, the following message appears:

```
Tuning up system configuration parameters
```

A few system parameters need to be raised to a set of higher values for better performance when using NFS. If the current values on your system are lower than those values (probably the case for the first-time installation), the installation procedure will prompt you to confirm that the values should be changed:

```
Tunable parameter "NQUEUE" is currently set to 256.

Is it O.K. to change it to 512? (y/n)
```

7. If you have at least 4 MB of memory on your system, type y to change the value; otherwise, type n. The system then asks:

```
Tunable parameter "NMUXLINK" is currently set to 32.

Is it O.K. to change it to 64? (y/n)
```

8. If you have at least 4 MB of memory on your system, type y to change the value; otherwise, type n. The system then asks:

```
Tunable parameter "NSTREAM" is currently set to 64.

Is it O.K. to change it to 96? (y/n)
```

9. If you have at least 4 MB of memory on your system, type y to change the value; otherwise, type n. The system then asks:

```
Tunable parameter "NBLK4096" is currently set to 4.

Is it O.K. to change it to 56? (y/n)
```

10. If you have at least 4 MB of memory on your system, type y to change the value; otherwise, type n. The system then asks:

```
Tunable parameter "NBLK2048" is currently set to 32.

Is it O.K. to change it to 80? (y/n)
```

11. If you have at least 4 MB of memory on your system, type y to change the value; otherwise, type n. The system then asks:

```
Tunable parameter "NBLK1024" is currently set to 32.

Is it O.K. to change it to 64? (y/n)
```

12. If you have at least 4 MB of memory on your system, type y to change the value; otherwise, type n. The system then asks:

```
Tunable parameter "NBLK512" is currently set to 32.

Is it O.K. to change it to 128? (y/n)
```

13. If you have at least 4 MB of memory on your system, type y to change the value; otherwise, type n. The system then asks:

```
Tunable parameter "NBLK256" is currently set to 64.

Is it O.K. to change it to 128? (y/n)
```

14. If you have at least 4 MB of memory on your system, type **y** to change the value; otherwise, type **n**. The system then asks:

```
Tunable parameter "NBLK128" is currently set to 128.

Is it O.K. to change it to 512 (y/n)
```

15. If you have at least 4 MB of memory on your system, it is recommended that you type **y** to change the value; otherwise, type **n**. The system then asks:

```
Tunable parameter "NBLK4" is currently set to 128.

Is it O.K. to change it to 256? (y/n)
```

16. If you have at least 4 MB of memory on your system, type **y** to change the value; otherwise, type **n**. Your screen will then look similar to this:

```
In order to complete the installation, a new kernel must be
built.
If you will later install a package
that builds a new kernel,
you may skip the kernel build.

Do you wish to build a new kernel now? [y, n]?
```

17. If you want to build a new kernel, type **y**. The **kconfig** utility will be called to build a new kernel. Your screen will look similar to this:

```
Building kernel to include Network File System Package.

Kernel rebuilt.

Installing a new kernel requires a system re-boot.
When finishing installing packages, to install the newly
built kernel, as user root, enter:

            cd /
            inskern unix.n

Run the system command 'setupnfs' first to initialize the
Network File System.  Use the sysadm command 'startstopnfs'
to start NFS.


Installation of INTERACTIVE NFS-Version 2.2 is complete.
```

Note that the *n* on the screen will be replaced by the appropriate kernel number for your system, based on the number of kernels you have built previously. If you do not want to build a new kernel now, type **n**. Your screen will look similar to this:

```
Run the system command 'setupnfs' first to initialize the
Network File System.  Use the sysadm command 'startstopnfs'
to start NFS.
```

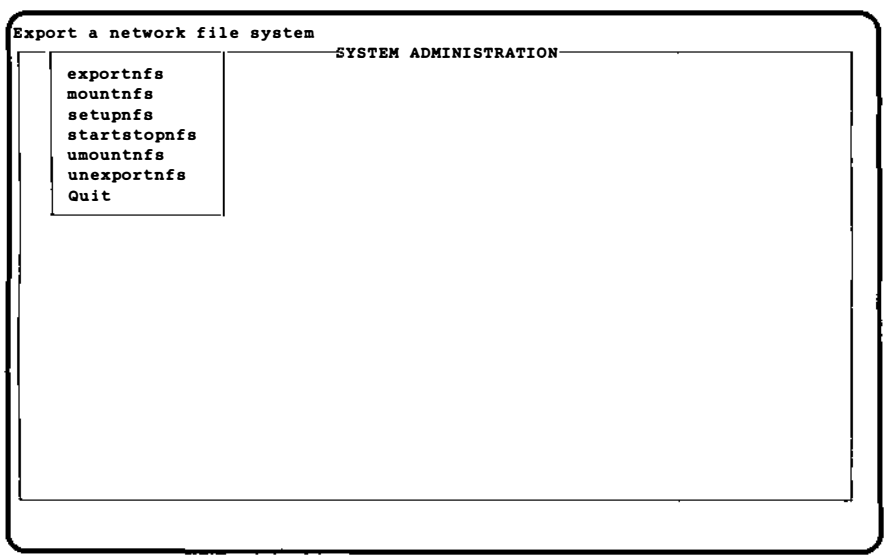
Follow these instructions after you have finished initializing INTERACTIVE NFS to install the new kernel.

The INTERACTIVE NFS extension is now installed and can be initialized for use. You will need to run `setupnfs` using the system administration `nfsmgmt` command. You will also need to build a new kernel that includes the INTERACTIVE NFS files you have just installed, if you did not do it when asked during installation.

5. THE NFS MANAGEMENT MENU

After you have installed the INTERACTIVE NFS extension on your fixed disk, initialize and maintain NFS on your system using the `sysadm nfsmgmt` (Network File System Management) command.

You can access the system administration bar menu by logging in as `sysadm` or by typing the `sysadm` command. As an alternative, you can type `sysadm nfsmgmt` to access the Network File System Management menu directly. Your screen will look similar to this:



Use the up and down arrow keys to move to the option you want, and press ENTER to select it.

Each option and its use is described in detail in section 6.

6. SETTING UP AND MAINTAINING INTERACTIVE NFS

6.1 Setting Up INTERACTIVE NFS (`setupnfs`)

The `setupnfs` option is used once only, at the time of INTERACTIVE NFS installation. The `setupnfs` option enables the INTERACTIVE NFS servers. (A *server* is a daemon process that performs actions requested by the client machines.)

To set up INTERACTIVE NFS, access the Network File System Management menu and select the `setupnfs` option. Your screen will look similar to this:

```
Enabling NFS Server
Enabling NFS Mount Server
Enabling Automatic Startup of NFS
```

6.2 Starting INTERACTIVE NFS (`startstopnfs`)

The `startstopnfs` option is used to start INTERACTIVE NFS sessions. The system must be at run level 3 for these commands to work.

1. Access the Network File System Management menu and select the `startstopnfs` option. Your screen will look similar to this:

```
Enter the item number of the operation you wish to use.
     1   Start NFS.
     2   Stop NFS.
     3   Start NFS and make entries to automatically start
         up NFS.
     4   Stop NFS and remove entries to automatically start
         up NFS.
     q   To exit.
Enter your selection:
```

2. Select option 1 if you want to start a single session of INTERACTIVE NFS. The screen will look similar to this:

```
Starting NFS
NFS Startup...
ONC daemons: portmap rexd pcnfsd rwalld
Lock manager: statd lckclnt(4) lockd
NFS daemons: nfsclnt(4) biod(4)
NFS Startup complete.
NFS initialized

Press the RETURN key to see the nfsmgmt menu [?, ^, q]:
```

3. If you want NFS to start automatically every time you bring the system up to multi-user/networking mode, select option 3 rather than option 1.

If you select option 3, your screen will look similar to this:

```
Starting NFS
NFS Startup...
ONC daemons: portmap rexd pcnfsd rwalld
Lock manager: statd lckclnt(4) lockd
NFS daemons: nfsclnt(4) biod(4)
NFS Startup complete.
Checking for files
Linking appropriate files
NFS initialized

Press the RETURN key to see the nfsmgmt menu [?, ^, q]:
```

The files necessary to run INTERACTIVE NFS are now in place.

6.3 Stopping INTERACTIVE NFS (`startstopnfs`)

At some time you may need to stop INTERACTIVE NFS, for example, if you are having network problems. The `startstopnfs` option is used to stop INTERACTIVE NFS sessions.

1. Access the Network File System Management menu and select the `startstopnfs` option. Your screen will look similar to this:

```
Enter the item number of the operation you wish to use.
     1  Start NFS.
     2  Stop NFS.
     3  Start NFS and make entries to automatically start
        up NFS.
     4  Stop NFS and remove entries to automatically start
        up NFS.
     q  To exit.
Enter your selection:
```

2. Select option 2 to simply stop the current session of INTERACTIVE NFS. Your screen will look similar to this:

```
Stopping NFS
NFS Shutdown: [NFS Shutdown Complete]
Press the RETURN key to see the nfsmgmt menu [?, ^, q]:
```

3. If you have previously configured the INTERACTIVE NFS extension to start automatically every time you bring the system up to multi-user/networking mode and you want to change this, select option 4 rather than option 2. INTERACTIVE NFS will no longer start every time you bring your system up to multi-user/networking mode; you will have to start it manually using the `startstopnfs` option. This operation may take up to 1 minute to complete. Your screen will look similar to this:

```
Stopping NFS
NFS Shutdown: [NFS shutdown complete]
Removing entries for automatically starting NFS

Press the RETURN key to see the nfsmgmt menu [?,q]:
```

6.4 Exporting a File System (`exportnfs`)

Exporting a file system notifies the other machines in the network of the availability of a file system on your local machine. Once a file system is exported, the system administrators of other machines in the network can mount that file system under the directories of their choice on their machines. Users on remote machines can then access files that exist in your local exported file system just as if they were physically located on the remote machines.

To export a file system, you provide the mount directory name of the file system you want to export and any restrictions you want to place on which *hosts* (machines) in the network may access the file system.

1. Start INTERACTIVE NFS (if it is not already running). Access the Network File System Management menu and select the `exportnfs` option. Your screen will look similar to this:

```
Enter the item number of the export operation you wish to
execute.  Options are:
        1  Export a file system.

        2  Add a host to the list of allowed hosts for a
           file system.

Select export operation (q):
```

2. To export a file system for the first time, select option 1. The system then asks:

```
Available file systems:
    /usr2
    /usr/local
    /v
    /usr

Enter the mount directory of the file system you want
to export [?, q]:
```

3. Enter the full mount directory name of the file system you plan to export, `/usr` for example. The system then asks:

```
Should /usr be exported read-only? [y, n, q]
```

4. Type `y` to prevent all remote users from altering `/usr` files and directories; type `n` if changes will be allowed. The system then asks:

```
Continue adding Host Names for /usr? [y, n, q]
```

5. Type `y` to add the name of a host that will be allowed to access `/usr`. Your screen will look similar to this:

```
Enter a Host name for /usr:
```

6. Type in the network node name of the machine (for example, `dorrit`) you want to give access to `/usr`. Your screen will then look similar to this:

```
Should dorrit be given root access? [y, n, q]
```

7. Type `y` to allow the `root` user access to all files on this file system, regardless of the files' permissions. Type `n` to deny this access.

   ☛ Note that selecting `y` could compromise your system's security. Select `y` *only* if you are sure you can trust the superuser(s) on the remote system to which you are exporting this file system.

   Your screen will then look similar to this:

```
Should dorrit be given read-only access? [y, n, q]
```

8. Type `y` to give `read-only` access to users on the remote host; type `n` to allow the remote users to modify files, if their permissions allow it. Note that `read-only` access only affects this particular client system. Your screen will then look similar to this:

```
Continue adding Host Names for /usr? [y, n, q]
```

9. Type `y` to repeat these steps for another machine; `n` to stop. The system will then ask for a comment or description of the file system:

```
Enter a brief comment about /usr:
```

10. Type in a short description, such as `Shared usr file system`. The system will then ask:

```
Okay to export /usr? [y, n, q]
```

11. Type `y` if you have entered all the necessary information correctly. The file system is then exported and your screen will look similar to this:

```
Press the RETURN key to see the nfsmgmt menu [?,q]:
```

6.5 Adding a Host to the List of Allowed Hosts (exportnfs)

1. Start INTERACTIVE NFS (if it is not already running). Access the Network File System Management menu and select the `exportnfs` option. Your screen will look similar to this:

```
Enter the item number of the export operation you wish to
execute.  Options are:
        1  Export a file system.

        2  Add a host to the list of allowed hosts for a
           file system.

Select export operation (q):
```

2. To add host machines for a file system you have already exported, select option 2. The system then asks:

```
Available file systems:

    /usr

Enter the mount directory of the file system you want
to add hosts for [?, q]:
```

3. Type in the name of the mount directory for `/usr`. The system then asks:

```
Current hosts already available for /usr:

    root allowed:   dorrit
    read/write:     dorrit muffy
    read-only:      rangoon

Continue adding Host Names for /usr? [y, n, q]
```

4. To add a new host, type `y`. Your screen will look similar to this:

```
Enter a Host name for /usr:
```

5. Type in the network node name of the machine (for example, `dorrit`) that you want to give access to `/usr`. Your screen will then look similar to this:

```
Should dorrit be given root access? [y, n, q]
```

6. Type `y` to allow the `root` user access to all files on this file system, regardless of the files' permissions. Type `n` to deny this access.

   ☛ Note that selecting `y` could compromise your system's security. Select `y` *only* if you are sure you can

trust the superuser(s) on the remote system to which you are exporting this file system.

Your screen will then look similar to this:

```
Should dorrit be given read-only access? [y, n, q]
```

7. Type `y` to give `read-only` access to users on the remote host; type `n` to allow the remote users to modify files, if their permissions allow it. Note that `read-only` access only affects this particular client system. Your screen will then look similar to this:

```
Continue adding Host Names for /usr? [y, n, q]
```

8. Type `y` to repeat these steps for another machine; `n` to stop. The system will then ask:

```
Okay to export /usr? [y, n, q]
```

9. Type `y` if you have entered all the necessary information correctly. The file system is then exported and your screen will look similar to this:

```
Press the RETURN key to see the nfsmgmt menu [?,q]:
```

6.6 Mounting a Remote File System or Changing Its Entry (`mountnfs`)

Mounting a remote file system under a directory on your machine lets users transparently access files and directories on that machine. All that is required to mount a remote file system is a valid directory under which it can be placed on the local machine.

1. Before beginning the mount procedure, identify or create a directory where the remote file system will be mounted.

2. Access the Network File System Management menu and select the `mountnfs` option. The screen will look similar to this:

```
Enter the item number of the mount operation you wish to
execute.  Options are:
      1  Mount a file system.

      2  Make an entry for automatically mounting a file
         system when NFS is started.

      3  Change an entry for automatically mounting a file
         system when NFS is started.

Select mount operation (q):
```

3. Select option 1 if you want the file system to be mounted during the current INTERACTIVE NFS session. Select option 2 if you want the file system to be mounted every time INTERACTIVE NFS is brought up. After selecting option 1, your screen will look similar to this:

```
Enter the Host you want to mount from:
```

4. Enter the name of the remote host system (for example, `dorrit`) that contains the file system that you want to mount on your local system. The screen will then look similar to this:

```
Available file systems:

    /v
    /y
    /etc
    /usr

Enter the file system you want to mount [q]:
```

5. Type `/usr` to select that file system for mounting. The system will then ask where it should be mounted:

```
Enter the directory to mount /usr under:
```

6. Enter the full path name of a valid directory under which `/usr` is to be mounted, for example, `/remote`. The system will then ask about restrictions:

```
Mount /usr as Read-Only? [y, n, q]
```

7. Type `y` if you do not want users on your local machine to have `write` permission in the remote file system. Type `n` if you want them to be able to modify the files. The screen will then look similar to this:

```
Mount /usr soft ? [y, n, q]
```

8. Type `y` if you want the system to return an error when the server does not respond. Type `n` if you want the system to continue to retry indefinitely. The screen will then look similar to this:

```
Enter the mount buffer size: [1) 1K, 2) 4K, 3) 8K]
```

9. Select an appropriate `read/write` buffer size. If there is no specific restriction on your LAN board (refer to the release notes that accompany this package), you should always select option 3 (8K) to get the best performance. The screen will then look similar to this:

```
OK to mount /usr under /remote? [y, n, q]
```

10. Type `y`. The system will then display the message:

```
/usr is now mounted under /remote.

Press the RETURN key to see the nfsmgmt menu [?, q]:
```

Option `2` is used to mount the file system automatically every time INTERACTIVE NFS is started in the future.

For example, to make an entry for mounting the file system `/usr` under `/remote`:

1. Access the Network File System Management menu and select `mountnfs`. Your screen will look similar to this:

```
Enter the item number of the mount operation you wish to
execute.  Options are:

        1. Mount a file system.

        2. Make an entry for automatically mounting a file
           system when NFS is started.

        3. Change an entry for automatically mounting a file
           system when NFS is started.

Select mount operation (q):
```

2. Select option `2`. The screen will then look similar to this:

```
Enter the Host you want to mount from:
```

3. Enter the name of the remote host system (for example, `dorrit`) that contains the file system that you want to mount on your local system. The screen will then look similar to this:

```
Available file systems:

    /v
    /y
    /etc
    /usr

Enter the file system you want to mount automatically [q]:
```

4. Type `/usr` to select that file system for mounting. The system will then ask about restrictions:

```
Automatically mount /usr read-only? [y, n, q]
```

5. Type `y` if you do not want users on your local machine to have `write` permission in the remote file system. Type `n` if

you want them to be able to modify the files. The screen will then look similar to this:

```
Automatically mount /usr soft ? [y, n, q]
```

6. Type `y` if you want the system to return an error when the server does not respond. Type `n` if you want the system to continue to retry indefinitely. The screen will then look similar to this:

```
Automatically mount /usr in the background? [y, n, q]
```

7. Type `y` if you want system startup to continue immediately while the mounting operation runs in the background rather than waiting for the mount to complete before startup proceeds. If you type `y`, and the server on which this file system is located is dead or hung, it will not delay the boot process on your machine. Type `n` if you do not want this protection. The screen will then look similar to this:

```
Enter the mount buffer size : [1)1K, 2)4K, 3)8K]
```

8. Select an appropriate `read/write` buffer size. If there is no specific restriction on your LAN board (refer to the release notes that accompany this package), you should always select option 3 (8K) to get the best performance. The system will then ask where it should be mounted:

```
Enter the directory to mount /usr under:
```

Enter the full path name of a valid directory under which `/usr` is to be mounted, for example, `/remote`. The system then asks:

```
Make entry for mounting /usr under /remote automatically
when NFS is started? [y, n, q]
```

9. Type `y`. The system then displays the message:

```
Entry made.

Mount dorrit:/usr under /remote now? [y, n, q]
```

10. Type `y` if you want to mount `/usr` immediately. Your screen will look similar to this:

```
/usr is now mounted under /remote.

Press the RETURN key to see the nfsmgmt menu [?, q]:
```

Option 3 is used to change entries for resources that are to be mounted automatically when INTERACTIVE NFS is started. You can change the file system name, the directory under which it is

mounted, and, if `write` permission is available on the file system, its permissions. If the resource is already mounted because an INTERACTIVE NFS session is currently running, the change will not take effect until INTERACTIVE NFS is stopped and started again. For example, to mount the file system `/usr` under `/client` instead of `/remote`, do the following:

1. Access the Network File System Management menu and select `mountnfs`. Your screen will look similar to this:

```
Enter the item number of the mount operation you wish to
execute.  Options are:
        1  Mount a file system.

        2  Make an entry for automatically mounting a file
           system when NFS is started.

        3  Change an entry for automatically mounting a
           file system when NFS is started.

Select mount operation (q):
```

2. Select option `3`. The screen will then look similar to this:

```
Current entries:
dorrit:/usr /remote -r NFS,rsize=4096,wsize=4096,soft,bg

Available File Systems:
dorrit:/usr
Enter the resource(s) you want to change [?, q]:
```

3. Type the name of the file system, for example, `dorrit:/usr`. The screen will then look similar to this:

```
Select:
        1       To change host/filesystem name
        2       To change directory name
        3       To turn on/off read-only
        q       To quit
Enter selection:
```

4. Type `2` to change the directory name. The system will then ask for a replacement name:

```
Enter replacement name for /remote
```

5. Type in the name of your new directory, for example, `/newremote`. The screen will then look similar to this:

```
Okay to change directory name /remote to
/newremote? [y, n, q]
```

6. Type `y` to change the directory. Your screen will then look similar to this:

```
/remote changed to /newremote

Press the RETURN key to see the nfsmgmt menu [?,q]:
```

6.7 Unexporting a File System (`unexportnfs`)

Unexporting a file system makes that file system on your local machine unavailable to remote users.

☛ Note that if you unexport a file system during an INTERACTIVE NFS session and that file system has already been mounted on another machine, the other machine will see error messages such as `Cannot statfs /usr: file descriptor in bad state.`

Before unexporting a file system, it is a good idea to type the command `showmount -a` to list the machines that have your exported file systems currently mounted. You may want to ask the system administrators of those machines to unmount your file system before you unexport it.

1. To unexport a file system, access the Network File System Management menu and select the `unexportnfs` option. Your screen will look similar to this:

```
Enter the item number of the export operation you wish to
execute.  Options are:

        1  Unexport a file system.

        2  Delete a host on the list of allowed hosts for
           a file system.

Select operation (q):
```

2. If you want to unexport a file system, select option 1. If you want to delete hosts from the list of hosts allowed to access a particular file system, select option 2. If you selected option 1, your screen will look similar to this:

```
Available file systems:
/usr
Enter the mount directory(s) of the file systems you want
to unexport [?, q]:
```

3. Enter the name of the file system, for example, `/usr`. If you selected option 1, the system then asks:

```
Okay to unexport /usr? [y, n, q]
```

4. Type `y` to unexport `/usr`.
5. If you selected option 2 above (delete a host on the list of allowed hosts), your screen will look similar to this:

```
Available file systems:
    /usr

Enter the mount directory(s) of the file systems
you want to delete hosts from [?, q]:
```

Type in the name of the mount directories you want. Your screen will look similar to this:

```
Current hosts available for deletion for /usr:

    root allowed:   dorrit
    read/write:     dorrit muffy kanga
    read-only:      rangoon

Enter Host name to delete for /usr:
```

6. Type in the remote host name. The system then asks:

```
Continue deleting host names for /usr? [y, n, q]
```

7. Type `y` if you want to delete additional hosts that are currently able to mount this file system. Type `n` if you do not want to delete any additional hosts. The system will then ask:

```
Okay to update /usr? [y, n, q]
```

8. Type `y` if you have entered all the necessary information correctly. The file system will then be unexported and your screen will look similar to this:

```
Press the RETURN key to see the nfsmgmt menu [?, q]:
```

6.8 Unmounting a Remote File System (`umountnfs`)

Unmounting a remote file system makes it unavailable to users on your local machine.

☛ Note that when you unmount a remote file system, local users must not be using that file system. If a user is currently in that file system, the attempt to unmount it will fail.

1. To unmount a remote file system, access the Network File System Management menu and select the `umountnfs` option. Your screen will look similar to this:

```
Enter the item number of the operation you wish to execute.
Options are:
        1  Unmount a file system.

        2  Remove an entry for automatically mounting a
           file system for when NFS is started.

Select operation (q):
```

2. Select option `1` if you want to unmount a file system for the current INTERACTIVE NFS session only. If the file system is

mounted automatically each time INTERACTIVE NFS is started, choosing option 1 will not affect its availability during future INTERACTIVE NFS sessions. Select option 2 if you want to permanently unmount a file system. If you select 1, your screen will look similar to this:

```
Mounted file systems:
/remote

Enter the file system(s) you want to unmount [?, q]:
```

3. Type in the mount directory name of the file system you want to unmount, for example, `/remote`. The system then asks:

```
Okay to unmount /remote [y, n, q]
```

4. Type `y`. The machine will then display:

```
/remote has been unmounted.

Press the RETURN key to see the nfsmgmt menu [?, q]:
```

Option 2 removes the entry in the file `/etc/fstab` that automatically mounts the selected file system.

1. If you select option 2, your screen will look similar to this:

```
Resources mounted automatically:
dorrit:/usr /remote  -r NFS,rsize=4096,wsize=4096,soft,bg

Enter Resources to remove:
```

2. Enter the remote file system name, in this case, `dorrit:/usr`. The screen then prompts for your confirmation:

```
Okay to remove permanent entry for dorrit:/usr? [y, n, q]
```

3. Type `y`. Your screen will look similar to this:

```
Entry for dorrit:/usr has been removed.

Press the RETURN key to see the nfsmgmt menu [?, q]:
```

GLOSSARY

exporting
: The method by which a file system is made available for sharing with remote machines.

host
: A machine in an NFS environment that is configured to share file systems.

server
: A daemon process that performs actions required by the client machines.

soft mount
: Return error if server does not respond.

Network Information Service
: An optional subset of the INTERACTIVE NFS package that allows the system administrator to maintain a common distributed database of system files (for example, `/etc/passwd`, `/etc/group`, and `/etc/hosts`) across all the machines on the network (formerly called the Yellow Pages subset).

INDEX

INTERACTIVE NFS
Protocol Specifications
and
User's Guide

CONTENTS

INTERACTIVE NFS*
Protocol Specifications
and
User's Guide
Release 3.2.5

1. NFS OVERVIEW

1.1 Introduction

This section provides an overview of the *INTERACTIVE Network File System* (NFS), which allows users to mount directories across the network and then to treat remote files as if they were local files. INTERACTIVE NFS is derived from System V NFS* developed by Lachman Associates, Inc. Advanced users may wish to skip this section and go on to read section 1.2, which shows examples of how NFS works. Casual users may not be interested in section 1.3, which discusses the NFS architecture. Section 1.4 provides a brief description of the other sections in this document.

The Network File System is a facility for sharing files in a heterogeneous environment of machines, operating systems, and networks. Sharing is accomplished by mounting a remote file system, then reading or writing files in place.

A distributed network of machines can provide more aggregate computing power than a mainframe computer, with far less variation in response time over the course of the day. Thus, a network is generally more cost-effective than a central mainframe. However, a mainframe has often been preferred for large programming projects and database applications because all files can be stored on a single machine.

Those who work with personal computers that are not part of a network know the inconveniences resulting from data fragmentation. Even in a network environment, sharing programs and data can sometimes be difficult. Either files have to be copied to each machine where they are needed or users have to log in to the remote machine that has the required files. Network logins are time-consuming, and having multiple copies of a file becomes confusing as incompatible changes are made to separate copies.

To solve this problem, a distributed file system that permits client systems to access shared files on a remote system has been designed. Client machines request resources provided by other machines, which are called servers. A server machine makes particular file systems available. Client machines can mount these as local file systems. Thus, users can access remote files as if they were on the local machine.

The INTERACTIVE NFS extension was not designed by extending the INTERACTIVE UNIX* Operating System onto the network; instead, it was designed to fit into the network services architecture. Thus, the INTERACTIVE NFS extension is not a distributed operating system, but rather an interface that allows a variety of machines and operating systems to play the role of client or server.

1.1.1 Computing Environments

The current computing environment in many businesses and universities looks like this:

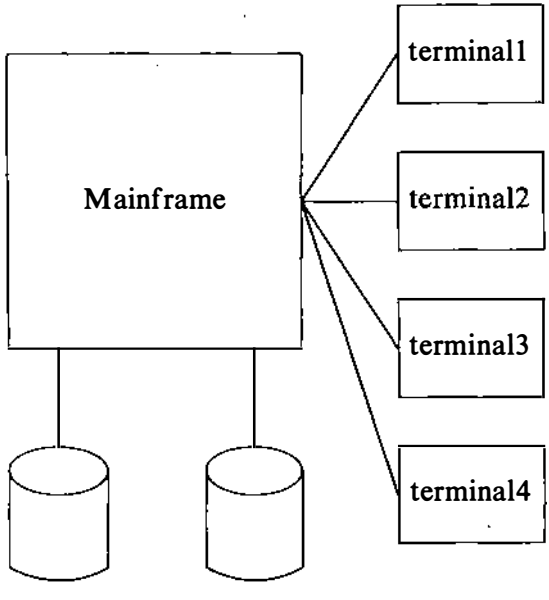


Figure 1. Typical Computing Environment

The major problem with this environment is competition for CPU cycles. A workstation environment solves that problem, but

introduces more disk drives into the picture. A network of workstations looks like this:

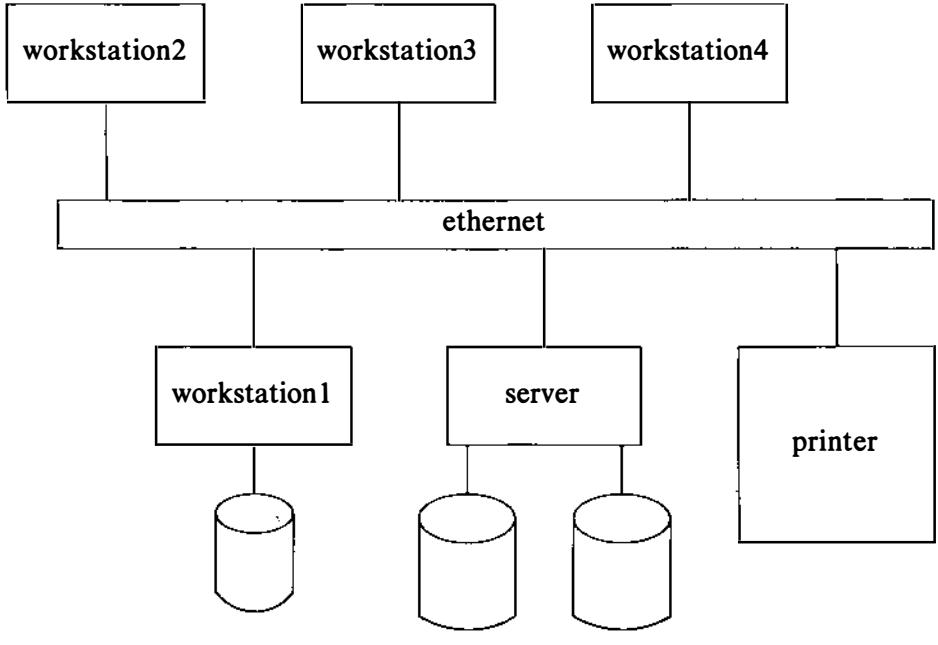


Figure 2. Typical Workstation Environment

The goal of NFS is to make all disks available as needed. Individual workstations have access to all information residing anywhere on the network. Printers and supercomputers may also be available on the network.

1.1.2 Terms and Concepts

A machine that provides resources to the network is a *server*, while a machine that employs these resources is a *client*. A machine may be both a server and a client. A person logged in on a client machine is a *user*, while a program or set of programs which runs on a client is an *application*. There is a distinction between the code implementing the operations of a file system (called *file system operations*) and the data making up the structure and contents of the file system (called *file system data*).

The *Remote Procedure Call* (RPC) facility provides a mechanism whereby one process (the *caller* process) can cause another process

(the *server* process) to execute a procedure call, as if the caller process had executed the procedure call in its own address space (as in the local model of a procedure call). Because the caller and the server are now two separate processes, they no longer need to exist on the same physical machine.

The RPC mechanism is implemented as a library of procedures, plus a specification for portable data transmission known as the *eXternal Data Representation* (XDR). Both RPC and XDR are portable, providing a standard I/O library for interprocess communication on one machine or across a network. Thus, programmers now have standardized access to these facilities without having to be concerned about the low-level details of any particular implementation.

The INTERACTIVE NFS extension is composed of a modified INTERACTIVE UNIX System kernel, a set of library routines, and a collection of utility commands. NFS presents a network client with a complete remote file system. Since NFS is largely transparent to the user, this document provides information on subjects of which an NFS user may be unaware. The INTERACTIVE NFS extension is an open system that can accommodate other machines on the net, even those not running the INTERACTIVE UNIX Operating System, without compromising security.

1.2 Examples of How INTERACTIVE NFS Works

1.2.1 Mounting a Remote File System

Suppose a user wants to read some on-line manual entries. These entries are not available on the client machine, called `client`, but are available on a machine called `docserv`. The directory containing the entries can be mounted as follows:

```
client# /etc/mount -f NFS docserv: /usr/man /usr/man
```

Note that the user must be `root` in order to issue a `mount` command. The `man` command can now be used whenever it is required; as it accesses the manual entries from the `/usr/man` directory, it transparently uses the copies located on the machine `docserv`. If the `df` command is run after the remote file system has been mounted, its output will look something like this:

```
/         (/dev/dsk/c1d0s0 ):     1636 blocks     1424 i-nodes
/usr      (/dev/dsk/c1d0s2 ):     5368 blocks     4480 i-nodes
/usr/man  (docserv:/usr/man):    36364 blocks        0 i-nodes
```

1.2.2 Exporting a File System

Suppose two users on different systems need to work together on a programming project. The source code is on the first user's machine, `senior`, in the directory `/usr/proj`.

Suppose that after creating the proper directory, the second user tries to remote mount the directory `/usr/proj`. Unless the directory has been specifically exported, the remote mount will fail with a "permission denied" message.

To export the directory, the first user must become superuser and edit the file `/etc/exports`. The following line should be placed in `/etc/exports`:

```
/usr/proj       -access=junior
```

if the second user is on a machine named `junior`. Without the specification of `-access=junior`, any system on the network would be permitted to remotely mount the directory `/usr/proj`. The *exportfs*(1M) command should then be run to export the file system just added to `/etc/exports`. The command will look like:

```
senior# /etc/exportfs -a
```

The `exportfs` command will also update the file, `/etc/xtab`. The NFS mount request server *mountd*(1M) will read the `/etc/xtab` file if necessary whenever it receives a request for a remote mount. Now the second user can remote mount the source directory by issuing this command:

```
junior# /etc/mount -f NFS senior:/usr/proj/usr/proj
```

1.2.3 Administering a Server Machine

System administrators must know how to set up the NFS server machine so that client workstations can mount all the necessary file systems. File systems are exported (that is, made available) by placing appropriate lines in the `/etc/exports` file. Here is a sample `/etc/exports` file for a typical server machine:

```
/
/usr
/usr/proj       -access=theteam
```

The path names specified in `/etc/exports` must be real file systems, that is, directory mount points for disk devices. The `root` file system must be exported if directories directly under `root`, such as `/lib` are to be made available to NFS clients. A

"netgroup," such as `theteam`, may be specified after the file system, in which case remote mounts are limited to machines that are members of this netgroup. Netgroups are defined in `/etc/netgroups` and are documented in *netgroup*(4). The complete format of `/etc/exports` and `/etc/xtab` files is described in *exports*(4). The *showmount*(1M) command shows which file systems have been exported.

1.2.4 Administering a Client Machine

The *mount*(1M) command is the tool for administering a client system. System administrators usually set up NFS on a client server machine so that users see all the necessary file systems already mounted by the time they log in. The file `/etc/fstab` is a convenient place to keep the associations between remote machine file systems and the local mount points. For example, in `/etc/fstab` something similar to the following may appear:

```
/dev/dsk/c1d0s2 /usr
docserv:/reference/rel3/usr/man /usr/man NFS
```

The second line enables the system administrator, or a user, to simply type:

```
client# /etc/mount /usr/man
```

and the remote association to machine `docserv` and its manual entries used will be made for the client system.

1.3 Architecture of NFS

1.3.1 Design Goals

1.3.1.1 Transparent Information Access. Users are able to get directly to the files they want without knowing the network address of the data. To the user, all universes look alike; there seems to be no difference between reading or writing a file contained on a private disk and reading or writing a file on a disk in the next building. Information on the network is truly distributed.

1.3.1.2 Different Machines and Operating Systems. The INTERACTIVE NFS extension is a standard for the exchange of data between different machines and operating systems.

1.3.1.3 Easily Extendable. A distributed system must have an architecture that allows integration of new software technologies without disturbing the existing software environment. To allow this, NFS provides network services, rather than a new network operating system. That is, NFS does not depend on extending the underlying

operating system onto the network, but instead offers a set of protocols for data exchange. These protocols can be easily extended.

1.3.1.4 Easy Network Administration. NFS has a convenient set of maintenance commands that have been developed over the years. Some new utilities are provided for network administration, but most of the old utilities have been retained.

The INTERACTIVE Network Information Service (NIS), formerly known as the Yellow Pages facility, is one example of a network service made possible with NFS. By storing password information and host addresses in a centralized database, the NIS facility eases the task of network administration.

The most obvious use of NIS is for administering `/etc/passwd`. Since NFS uses a UNIX System protection scheme across the network, it is advantageous to have a common `/etc/passwd` database for all machines on the network. NIS allows a single point of administration and gives all machines access to a recent version of the data.

Since the NIS interface is implemented using RPC and XDR, the service is available to non-UNIX operating systems. NIS servers do not interpret data, so it is possible for new databases to take advantage of the NIS service without modifying the servers. For more information on the INTERACTIVE Network Information Service, see the *INTERACTIVE Network Information Service Guide.*

1.3.1.5 Reliable. The file server protocol is designed so that client workstations can continue to operate even when the server crashes and reboots. Continuation after reboot is achieved without making assumptions about the fail-stop nature of the underlying server hardware.

The major advantage of a stateless server is robustness in the face of client, server, or network failures. Should a client fail, it is not necessary for a server (or human administrator) to take any action to continue normal operation. Should a server or the network fail, it is only necessary that clients continue to attempt to complete NFS operations until the server or network returns to the net. This robustness is especially important in a complex network of heterogeneous systems, many of which are not under the control of a disciplined operations staff, and which may be running untested systems often rebooted without warning.

1.3.1.6 High Performance. The flexibility of NFS allows configuration for a variety of cost and performance trade-offs. For example, configuring servers with large, high-performance disks and clients with no disks may yield better performance at lower cost than having many machines with small, inexpensive disks. Furthermore, it is possible to distribute the file system data across many servers and achieve the added benefit of multi-processing without losing transparency. In the case of read-only files, copies can be kept on several servers to avoid bottlenecks.

Several performance enhancements have been made to the INTERACTIVE NFS extension, such as "fast paths" for frequent operations, asynchronous service of multiple requests, caching of disk blocks, and asynchronous read-ahead and write-behind. The fact that caching and read-ahead occur on both client and server machines effectively increases the cache size and read-ahead distance. Caching and read-ahead do not add state to the server; nothing (except performance) is lost if cached information is thrown away. In the case of write-behind, both the client and server attempt to flush critical information to disk whenever necessary to reduce the impact of an unanticipated failure; clients do not free write-behind blocks until the server verifies that the data is written.

1.3.2 INTERACTIVE NFS Implementation

In the INTERACTIVE UNIX System implementation of NFS, three entities must be considered: the operating system interface, the logical file system (FSS) interface, and the Network File System (NFS) interface. The INTERACTIVE UNIX Operating System interface has not been modified for the implementation of NFS, ensuring compatibility with existing applications.

The FSS interface is a set of routines within the INTERACTIVE UNIX Operating System kernel that separates file system operations from the semantics of their implementation. Above the FSS interface, the operating system deals with generic file objects; below this interface, the file system specific data structures implement a file object.

The remote file system defines and implements the NFS interface using the RPC mechanism. RPC allows communication with remote services in a manner similar to the procedure calling mechanism available in many programming languages. The RPC protocols are described using the XDR package. XDR permits a machine-

independent representation and definition of high-level protocols on the network.

The following figure shows the flow of a request from a client (at the top left) to a server machine where the data requested lives on a local disk. The boxes labeled "filesys" refer to those functions within the operating system that provide access to data on the system's attached disks.

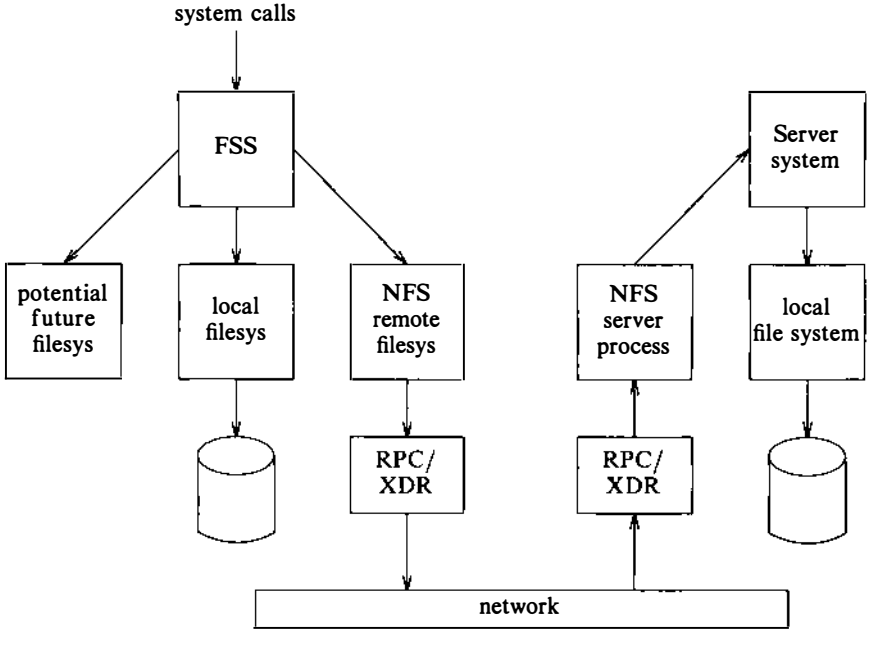


Figure 3. Flow of Request From Client to Server

In the case of access through a local file system, requests are directed to file system data on devices connected to the client machine. In the case of access through a remote file system, the request is passed through the RPC and XDR layers onto the network. In the current implementation, UDP/IP protocols and the Ethernet* are used. On the server side, requests are passed through

the RPC and XDR layers to an NFS server. This path is retraced to return results.

The INTERACTIVE UNIX System implementation of NFS provides five types of transparency:

1. *File System Type:* The FSS permits an operating system to interface transparently to a variety of file system types.
2. *File System Location:* Since there is no differentiation in the interface to a local or a remote file system implementation, the location of file system data is transparent.
3. *Operating System Type:* The RPC mechanism allows interconnection of a variety of operating systems on the network and makes the operating system type of a remote server transparent.
4. *Machine Type:* The XDR definition facility allows a variety of machines to communicate on the network and makes the machine type of a remote server transparent.
5. *Network Type:* RPC and XDR can be implemented for a variety of network and internet protocols, thereby making the network type transparent.

Other NFS implementations are possible at the expense of some advantages of the INTERACTIVE UNIX System version. In particular, a client (or server) may be added to the network by implementing one side of the NFS interface. An advantage of the INTERACTIVE UNIX System implementation is that the client and server sides are identical; thus, it is possible for any machine to be client, server or both. Users at client machines with disks can arrange to share over the NFS without having to appeal to a system administrator or configure a different system on their workstation.

1.3.3 The NFS Interface

The NFS interface itself is open and can be used by anyone wishing to implement an NFS client or server for the network. The interface defines traditional file system operations for reading directories, creating and destroying files, reading and writing files, and reading and setting file attributes. The interface is designed so that file operations address files with an uninterpreted identifier, starting byte address, and length in bytes.

Commands are provided for NFS servers to initiate service (`mountd`) and to serve a portion of their file system to the network (`/etc/exports`). A client builds its view of the file systems available on the network with the `mount` command.

The NFS interface is defined so that a server can be *stateless*. This means that a server does not have to remember anything about its clients from one transaction to the next, transactions completed, or files operated on. For example, there is no `open` operation, as this would imply state in the server; of course, the INTERACTIVE UNIX Operating System interface uses an `open` operation, but the information in the INTERACTIVE UNIX System operation is remembered by the client for use in later NFS operations.

The stateless nature of an NFS server causes a problem when an INTERACTIVE UNIX System application `unlinks` an open file. This is done to achieve the effect of a temporary file that is automatically removed when the application terminates. If the file in question is served by NFS, the `unlink` removes the file, since the server does not remember that the file is open. Thus, subsequent operations on the file will fail. In order to avoid state on the server, the client operating system detects the situation, renames the file rather than unlinking it, and unlinks the file when the application terminates. In certain failure cases, this leaves unwanted "temporary" files on the server; these files can be removed as a part of periodic file system maintenance.

Another example of how NFS provides a friendly interface to the INTERACTIVE UNIX Operating System without introducing state is the `mount` command. A client of NFS "builds" its view of the file system on its local devices using the `mount` command; thus, it is natural for the client to initiate its contact with NFS and build its view of the file system on the network via an extended `mount` command. This `mount` command does not imply state in the server, since it only acquires information for the client to establish contact with a server. The `mount` command may be issued at any time, but is typically executed as a part of client initialization. The corresponding `umount` command is only an informative message to the server, but it does change state in the client by modifying its view of the file system on the network.

The major advantage of a stateless server is robustness in the face of client, server, or network failures. Should a client fail, it is not necessary for a server (or human administrator) to take any action

to continue normal operation. Should a server or the network fail, it is only necessary that clients continue to attempt to complete NFS operations until the server or network is fixed. This robustness is especially important in a complex network of heterogeneous systems, many of which are not under the control of a disciplined operations staff and may be running untested systems and/or may be rebooted without warning.

An NFS server can be a client of another NFS server. However, a server does not act as an intermediary between a client and another server. Instead, a client may ask what remote mounts the server has and then attempt to make similar remote mounts. The decision to disallow intermediary servers is based on several factors. First, the existence of an intermediary will impact the performance characteristics of the system; the potential performance implications are so complex that it seems best to require direct communication between a client and server. Second, the existence of an intermediary complicates access control; it is much simpler to require a client and server to establish direct agreements for service. Finally, disallowing intermediaries prevents circularity in the service arrangements; this is preferred to detection or avoidance schemes.

NFS currently implements file protection by making use of the authentication mechanisms built into RPC. This retains transparency for clients and applications that make use of file protection. Although the RPC definition allows other authentication schemes, their use may have adverse effects on transparency.

Although NFS is UNIX System-friendly, it does not support all UNIX System file system operations. For example, the "special file" abstraction of devices is not supported for remote file systems because it is felt that the interface to devices would greatly complicate the NFS interface. Other incompatibilities are due to the fact that NFS servers are stateless. For example, file locking and guaranteed `append_mode` are not supported for the remote case by NFS. These services are provided through other, transparent network services.

Omitting certain features from NFS preserves the stateless implementation of servers and defines a simple, general interface. The availability of open RPC and NFS interfaces means that customers and users who need stateful or complex features can implement them "beside" or "within" NFS.

1.4 Network Documentation Roadmap

Section 2, "NFS ADMINISTRATION," is a reference guide for system administrators implementing and administering the Network File System on new machines.

Section 3, "RPCGEN PROTOCOL COMPILER," is intended for programmers who wish to write RPC applications simply and directly, instead of spending most of their time debugging their network interface code.

Section 4, "RPC GUIDE," is intended for programmers who wish to write network applications using remote procedure calls, thus avoiding low-level system primitives. Readers must be familiar with the C programming language and should have a working knowledge of network theory.

Section 5, "RPC PROTOCOL SPECIFICATION," is a reference guide for system programmers implementing the Network File System on new machines. It is of little interest to programmers writing network applications.

Section 6, "XDR PROTOCOL SPECIFICATION," is intended for programmers who write complicated applications using remote procedure calls and who need to pass complicated data across the network. It is also a reference guide for system programmers implementing the Network File System on new machines.

Section 7, "NFS PROTOCOL SPECIFICATION," is a reference guide for system programmers implementing the Network File System on new machines. It is of little interest to programmers writing network applications.

Section 8, "AUTOMOUNTER GUIDE," is a reference guide for system administrators implementing and administering the System V Network File System, using the automounter facility for NFS.

2. NFS ADMINISTRATION

2.1 Introduction

This section introduces the NFS service. The service currently available is described, and some terms in the network environment are defined.

Following that, the NFS service is explained, and information about periodic maintenance and trouble-shooting for this service can be found.

While some of this material tends to be theoretical, its specific implications will be seen again and again as you become familiar with system administration. For example, a user running NFS must understand that some typical INTERACTIVE UNIX Operating System procedures have changed in the NFS environment. This section covers only those aspects of the NFS network service necessary for performing the duties of system administration.

2.1.1 Terminology

Any machine that provides a network service is a *server.* Servers are entirely passive and wait for clients to call them; they never call the clients.

A *client* is any entity that accesses a network service. The term entity is used because the thing doing the accessing may be an actual machine or simply an INTERACTIVE UNIX System process generated by a piece of software.

The degree to which clients are bound to their servers varies with each of the network services. For example, an NIS client binds randomly to one of the NIS servers by broadcasting a request. At any point, the NIS client may decide to broadcast for a new server. However, an NFS client selects a specific server from which to mount a given file system.

The client initiates the binding. The server completes the binding subject to access control rules. Since most network administration problems occur at bind time, a system administrator should know how a client binds to a server and what (if any) access control policy the server uses.

2.1.2 The INTERACTIVE UNIX Operating System and the NFS Network Service

Unlike many recently marketed distributed operating systems, the UNIX operating system was originally designed without knowledge that networks existed. This "networking ignorance" presents three impediments to linking the INTERACTIVE UNIX Operating System with currently available high performance networks:

1. The UNIX System was never designed to yield to a higher authority, such as a network authentication server, for critical information or services. As a result some INTERACTIVE UNIX Operating System semantics are hard to maintain "over the net." For example, it may not always be appropriate to trust user ID 0 (`root`).

2. Some INTERACTIVE UNIX Operating System execution semantics are difficult. For example, the INTERACTIVE UNIX Operating System allows a user to remove an open file, yet the file does not disappear until closed by everyone. In a network environment a client INTERACTIVE UNIX System machine may not own an open file. Therefore, a server may remove a client's open file.

3. When an INTERACTIVE UNIX Operating System machine crashes, it takes all its applications down with it. When a network node crashes, whether client or server, it should not drag all of its bound neighbors down. The treatment of node failure on a network raises difficulties in any system and is especially difficult in the UNIX System environment. A system of "stateless" protocols has been implemented to circumvent the problem of a crashing server dragging down its bound clients. Stateless here means that a client is independently responsible for completing work and that a server need not remember anything from one call to the next. In other words, the server keeps no state. With no state left on the server, there is no state to recover when the server crashes and comes back up. From the client's point of view, a crashed server appears to be no different than a very slow server.

In implementing the INTERACTIVE UNIX Operating System over the network, NFS remains compatible with the INTERACTIVE UNIX Operating System whenever possible. However, certain incompatibilities have been introduced. These are typically of two kinds: first, those issues that would make a networked

INTERACTIVE UNIX System evolve into a distributed operating system, rather than a collection of network services, and second, those issues that would make crash recovery extremely difficult from both the implementation and administration point of view.

All incompatibilities are documented in the appropriate administration sections.

2.1.3 Debugging the INTERACTIVE UNIX Operating System in the Network Environment

Most problems involving the NFS network service lie in the one of the following four areas, which are listed in order of probability:

1. The network access control policies do not allow the operation, or architectural constraints prevent the operation.
2. The client software or environment is broken.
3. The server software or environment is broken.
4. The network is broken.

The following sections present specific instructions on how to check for these causes of failure in the NFS environment.

2.2 NFS: The Network File System

2.2.1 What Is the NFS Service?

NFS enables users to share file systems over the network. A client may mount or unmount file systems from an NFS server machine. The client always initiates the binding to a server's file system by using the *mount*(1M) command. Typically, a client remembers specific remote file systems and their mount points by placing lines like these in the file `/etc/fstab`:

```
titan:/usr2 /usr2 NFS
venus:/usr/man /usr/man -r NFS,soft
```

See *fstab*(4) for a full description of the format.

Since clients initiate all remote mounts, NFS servers keep control over who may mount a file system by limiting named file systems to desired clients with an entry in the `/etc/exports` file. For example:

```
/usr/local                          # export to any machine
/usr2   -access=bigmo:larry:curley  # export to only these machines
```

Note that path names given in `/etc/exports` must be the mount point of a local file system. See *exports*(4) for a full description of the format.

2.2.2 How NFS Works

Two remote programs implement the NFS service – *mountd*(1M) and *nfsd*(1M). A client's `mount` request talks to `mountd` which checks the access permission of the client and returns a pointer to a file system. After the `mount` completes, access to that mount point and below goes through the pointer to the server's `nfsd` daemon using *rpc*(4). Client kernel file access requests (write-behind and read-ahead) are handled by the *biod*(1M) daemons on the client.

2.2.3 Becoming an NFS Server

An NFS server is simply a machine that exports a file system or systems. The following steps must be taken to enable any machine to export a file system:

1. The superuser must place the mount point path name of the file system to be exported in the file `/etc/exports`. See *exports*(4) for file format details. For example, to export `/usr/lbin`, the export file would look like:

   ```
   /usr/lbin
   ```

   Of course, an NFS server may only export file systems of its own.

2. The entries in `/etc/exports` need to be exported. They can be exported with the *exportfs*(1M) command. The `exportfs` command is usually run from the NFS startup script, usually `/etc/rc3.d/s72nfs`.

3. The `/etc/mountd` program must be running for a remote mount to succeed. This is started from the NFS startup script, `/etc/rc3.d/s72nfs`.

4. A remote mount also needs some number of `nfsd` NFS daemon processes to be running on the NFS server. The actual number depends on the number of client NFS requests that the server should be able to handle concurrently, and thus depends on the speed and capacity of the server machine. This example shows four `nfsd` daemons. The NFS startup script, such as `/etc/rc3.d/s72nfs`, should be checked for lines like these:

```
if [ -f /etc/exports ]
then
        echo " nfsd(x4)\c"
        nfsd 4
fi
```

These lines (or similar ones) should be added to the new NFS server's NFS startup script to enable `nfsd`s. The superuser can enable the `nfsd`s daemons at any time by typing:

```
# /etc/nfsd 4
```

After these steps, the NFS server should be able to export the named file system.

2.2.4 Remote Mounting a File System

Any exported file system can be remote mounted onto a machine, as long as its server can be reached over the network and the machine is included in the `/etc/exports` list for that file system. On the machine where the file system is to be mounted, the superuser should type the following:

```
# mount -f NFS server_name:/file_system /mount_point
```

For example, to mount the manual entries from remote machine `elvis` on the directory `/usr/elvis.man`, type:

```
# mount -f NFS elvis:/usr/man /usr/elvis.man
```

To make sure the file system is mounted where it is expected to be, use the *mount*(1M) command without any arguments. This displays the currently mounted file systems.

Frequently used file systems should be listed, with any needed options, in the file `/etc/fstab`. See *fstab*(4) for the syntax and contents of the file.

2.2.5 Debugging NFS

Before trying to debug NFS, the user should read the section on how NFS works and also the manual entries for *mount*(1M), *mountd*(1M), *nfsd*(1M), *rpcinfo*(1M), *showmount*(1M), *exports*(4), *fstab*(4), *mnttab*(4), and *rmtab*(4). It is not necessary to understand them fully, but the user should be familiar with the names and functions of the various daemons and database files.

When tracking down an NFS problem, the user should keep in mind that, like all network services, there are three main points of failure: the server, the client, or the network itself. The debugging strategy outlined below tries to isolate each component to find the one that is

not working. For example, consider a sample `mount` request as made from an NFS client machine:

```
$ mount -f NFS krypton:/usr/src /krypton.src
```

The example asks the server machine `krypton` to return a file handle (`fhandle`) for the directory `/usr/src`. This `fhandle` is then passed to the kernel in the *mount*(2) system call. The kernel looks up the directory `/krypton.src`, and if everything is correct, it ties the `fhandle` to the directory in a mount record. From now on all file system requests to that directory and below will go through the `fhandle` to the server `krypton`.

The above describes the way in which the system should work. Be aware that some things may go wrong. Following are some general pointers and then a list of the possible errors and what might have caused them.

2.2.5.1 General Hints. When there are network or server problems, programs that access hard mounted remote files fail in different ways to those which access soft mounted remote files. Hard mounted remote file systems cause programs to retry until the server responds again. Soft mounted remote file systems return an error after trying for awhile.

Once a hard mount succeeds, programs that access hard mounted files hang as long as the server fails to respond. In this case, NFS should print a "server not responding" message on the console. On a soft mounted file system programs get an error when a file whose server is dead is accessed.

If a client is having NFS trouble, the first check must be to make sure the server is up and running. From a client, the following should be typed to see if the server is up at all:

```
$ rpcinfo -p server_name
```

It should print out a list of program, version, protocol, and port numbers that resembles:

```
program  vers proto   port
 100000    2   tcp    111  portmapper
 100000    2   udp    111  portmapper
 100017    1   tcp   1024  rexd
 100005    1   udp   1027  mountd
 100003    2   udp   2049  nfs
 100024    1   udp   1039  status
 100024    1   tcp   1025  status
 100021    1   tcp   1026  nlockmgr
 100021    1   udp   1051  nlockmgr
 100020    1   udp   1054  llockmgr
 100020    1   tcp   1028  llockmgr
 100021    2   tcp   1032  nlockmgr
 100004    2   udp   1063  ypserv
 100004    2   tcp   1033  ypserv
 100004    1   udp   1063  ypserv
 100004    1   tcp   1033  ypserv
 100007    2   tcp   1034  ypbind
 100007    2   udp   1074  ypbind
 100007    1   tcp   1034  ypbind
 100007    1   udp   1074  ypbind
```

The `rpcinfo` command can also be used to check if the `mountd` server is running:

```
$ rpcinfo -u server_name mountd
```

This should return:

```
program 100005 version 1 ready and waiting
```

If these steps fail, a login should be tried on the server's console to see if it is running.

If the server is alive but a client machine cannot reach it, the Ethernet connections between the machines should be checked.

If the server and the network are alive, use `ps` to check the client daemons. An `nfsclnt` and several `biod` daemons should be running. For example:

```
$ ps -ef
```

should print lines for `nfsclnt` and `biod`.

The four sections below deal with the most common types of failure. The first covers the steps to be taken if a remote mount fails; the next three discuss servers that do not respond when file systems are mounted.

2.2.5.2 Remote Mount Failed. This section deals with problems related to mounting. If `mount` fails for any reason, the sections below should be checked for specific details about what to do. They

are arranged according to where they occur in the mounting sequence and are labeled with the error message likely to be displayed.

The `mount` command can get its parameters either from the command line or from the file `/etc/fstab` (see *mount*(1M)). The example below assumes command line arguments, but the same debugging techniques would apply if `/etc/fstab` was the source of the options.

The interaction of the various parts of the `mount` request should be considered. In the example `mount` request given above:

```
$ mount -f NFS krypton:/usr/src /krypton.src
```

`mount` goes through the following steps to mount a remote file system.

1. The `mount` command opens `/etc/mnttab` and checks that this mount has not already been done.
2. The `mount` command parses the first argument into host `krypton` and remote directory `/usr/src`.
3. The `mount` command then resolves the host, `krypton`, into an internet protocol address.
4. The `mount` command calls `krypton`'s portmapper to get the port number of `mountd`.
5. The `mount` command calls `krypton`'s `mountd` and passes it to `/usr/src`.
6. `krypton`'s `mountd` reads `/etc/exports` and looks for the exported file system that contains `/usr/src`.
7. `krypton`'s `mountd` may call the NIS server `ypserv` to expand the host names and netgroups in the export list for `/usr/src`.
8. `krypton`'s `mountd` does a *getfh*(2) system call on `/usr/src` to get the `fhandle`.
9. `krypton`'s `mountd` returns the `fhandle`.
10. Back on the client, `mount` does a *mount*(2) system call with the `fhandle` and `/krypton.src`.
11. The `mount` command checks if the caller is superuser and if `/krypton.src` is a directory.

12. The `mount` command does a *statfs*(2) call to `krypton`'s NFS server (`nfsd`).

13. The `mount` command opens `/etc/mnttab` and adds an entry.

Any one of these steps can fail, some of them in more than one way. The entries below give detailed descriptions of the failures associated with specific error messages:

`mount: cannot open /etc/mnttab`
: The table of mounted file systems is kept in the file `/etc/mnttab`. This file must exist before `mount` can succeed. `/etc/mnttab` is created when the system is booted and is maintained automatically after that by the `mount` and `umount` commands.

`mount: /dev/nfsd is already mounted, ... is busy, or allowable number of mount points exceeded`
: This message reveals an attempt to mount a file system which is already mounted. All NFS `mount` requests that fail with this message will display the name `/dev/nfsd` (a byproduct of the implementation) regardless of the actual `mount` request.

`mount: ... or ..., no such file or directory`
: The `NFS` or `krypton:` part of

```
# mount -f NFS krypton:/usr/src /krypton.src
```

was probably omitted. The `mount` command assumes that a local mount is being done unless the `-f` flag is used on the command line, or the requested directory as listed in `/etc/fstab` specifies file system type `NFS`.

More simply, this message also appears when for a correct `mount` request, the specified local mount point is not an existing directory.

`mount: cannot open </etc/fstab>`

The `mount` command tried to look up the information needed to complete a `mount` request in `/etc/fstab`, but there was no such file. This file should be created by the system administrator as part of initial system setup.

`... not in hosts database`

The system name specified on the `mount` request suffixed by the colon (:) could not be resolved to a network address. The spelling of the host name and placement of the colon in the `mount` call should be checked.

`mount: directory argument <...> must be a full path name`

The second argument to `mount` is the path of the directory to be covered. This must be an absolute path starting at /.

`mount: ... server not responding(1): RPC_PMAP_FAILURE - RPC_TIMED_OUT`

Either the server to which the `mount` is being attempted is down or its portmapper is dead or hung. An attempt should be made to log in to that machine; if that attempt succeeds, the problem may be in the portmapper. Run the following from your system to test the portmapper on the server system:

```
# rpcinfo -p hostname
```

The result should be a list of registered programs. If this is not the case, the remote portmapper must be killed and re-started. Restarting the portmapper is a complicated process because all registered services are lost, and their associated daemons must be restarted also. This is done by the superuser as follows:

```
#  ps -ef
```

find the process IDs of `portmap` and other service daemons.

```
# kill -9 portmap_pid daemon_id1 daemon_id2
```

is used to kill the daemons; then, as an example:

```
# /etc/portmap
# /etc/mountd
# /etc/nfsd 4
#  and so on
```

to start new ones.

Another alternative to all this is to simply reboot the server when it is convenient. Because of the stateless nature of the NFS server implementation, there should be no adverse effect on the clients of the system other than the time that they will suspend awaiting the return of the server.

If the server is up but it is not possible to `rlogin` to it, the client's Ethernet connection should be checked by trying to `rlogin` to some other machine. The server's Ethernet connection should also be checked.

`mount: ... server not responding: RPC_PROG_NOT_REGISTERED`

This means that `mount` got through to the portmapper, but the NFS mount daemon `mountd` was not registered. The server should be checked to ensure that `/etc/mountd` exists and is running.

`mount: /dev/nfsd or ..., no such file or directory`

Either the remote directory does not exist on the server or the local directory does not exist. Again note that `/dev/nfsd` will always be printed to represent the remote directory.

`mount: access denied for ...: ...`

The machine on which the `mount` attempt is being made is not in the server's export list for the file system to be mounted. A list of the server's exported file systems can be obtained by running:

```
# showmount -e hostname
```

If the file system which is wanted is not in the list or the machine name or netgroup name is not in the user list for the file system, the `/etc/exports` file on the server should be checked for the correct file system entry. A file system name that appears in the `/etc/exports` file but not in the output from `showmount` indicates a failure in `mountd`. Either it could not parse that line in the file, it could not find the file system, or the file system name was not a local mounted file system. See *exports*(4) for more information.

This message can also be an indication that authentication failed on the server. It may be displayed because the machine that is attempting the `mount` is not in the server's export list, because the server is not aware of the machine, or because the

server does not believe the identity of the machine. The server's `/etc/exports` file should be checked.

Another possible authentication problem may be that the client process doing the `mount` request may be a member of too many groups for the RPC on the server to handle. Various servers can have different numbers of simultaneous groups. The limit should be checked on the server.

`mount: ... not a directory`
The remote path on the server is not a directory.

`mount: not super user`
The `mount` command can be done only by the superuser because it affects the file system for the whole machine.

2.2.5.3 Programs Hung. If programs hang doing file related work, the NFS server may be dead. The message:

```
WARNING: NFS server sysname not responding, still trying
```

may be displayed on the machine's console. The message includes the name of the NFS server which is down.

This is probably a problem either with one of the NFS servers or with the Ethernet.

If a machine hangs completely, a check should be made of the server(s) from which file systems have been mounted. If one (or more) of them is down, client machines may hang. When the server comes back up, programs will continue automatically and will not be affected.

If a soft mounted server dies, other work should not be affected. Programs that time-out trying to access soft mounted remote files will fail, but it should still be possible to get work done on other file systems.

If other clients of the server seem to be functioning correctly, the Ethernet connection and connection of the server should be checked.

2.2.5.4 Everything Works Slowly. If access to remote files seems unusually slow, the server should be checked by entering (on the server):

```
# ps -ef
```

If the server is functioning and other users are getting good response, block I/O daemons on the client should be checked by

typing `ps -ef` (on the client) and looking for `biod`. If the daemons are not running or are hung, they should be killed by typing:

```
# ps -ef | grep biod
```

to find the process IDs, followed by:

```
# kill -9 pid1 pid2 pid3 pid4
```

The daemons should then be restarted with:

```
# /etc/biod 4
```

To determine whether the daemons are hung, `ps` should be used as above, then a large file copied. Another `ps` will show whether the `biod`s are accumulating cpu time: if not, they are probably hung.

If `biod` appears to be functioning correctly, the Ethernet connection should be checked. The `nfsstat -c` and `nfsstat -s` commands can be used to discover whether a client is doing a lot of retransmitting. A retransmission rate of 5 percent is considered high. Excessive retransmission usually indicates a bad Ethernet board, a bad Ethernet tap, a mismatch between board and tap, or a mismatch between the client machine's Ethernet board and the server's board.

It is also possible that the server's Ethernet board can not handle the load being placed on it. In this case, remote file systems from this server should be unmounted and then remounted with reduced `read` and `write` sizes. This will help to avoid IP fragmentation and will reduce the number of back-to-back packets that the server will need to be able to handle. See the NFS mount options as described in *mount*(1M) for more information. Typically, reducing both sizes to 4096 or even 1024, will help.

In rare cases, it is possible for the client's Ethernet interface to be unable to handle the load that the client is generating. The solution is the same in this case, reducing the `read` and `write` sizes.

2.2.6 Incompatibilities With Standard INTERACTIVE UNIX System Releases

A few things work in different ways, or not at all, on remote NFS file systems. The next section discusses the incompatibilities and offers suggestions on working around them.

2.2.6.1 No SU Over the Network. Under NFS a server exports file systems it owns so that clients may remote mount them. When a client becomes superuser, it may be denied permission on remote mounted file systems, depending on the `exportfs` options. Remote `su (root)` access to files may be enabled by changing `/etc/exports` on the server to permit `root` access. See *exportfs*(1M) and *exports*(4) for details. Consider the following example:

```
$  touch test1 test2
$  chmod 777 test1
$  chmod 700 test2
$  ls -l test*
-rwxrwxrwx  1 jsbach       0 Mar 24 16:12 test1
-rwx------  1 jsbach       0 Mar 24 16:12 test2
```

The example is tried again by the superuser:

```
$  su
Password:
#  touch test1
#  touch test2
touch: test2: Permission denied
#  ls -l test*
-rwxrwxrwx  1 jsbach       0 Mar 24 16:16 test1
-rwx------  1 jsbach       0 Mar 24 16:12 test2
```

The problem usually shows up during the execution of a set-uid `root` program. Programs that run as `root` cannot access files or directories unless the permission for "other" allows it.

Another aspect of this problem is that ownership of remote mounted files cannot always be changed, specifically, if they are on a server that does not permit users to execute `chown`. Since `root` is treated as the "other" user for remote accesses, only `root` on the server can change the ownership of remote files. For example, consider a user trying to `chown` a new program, `a.out`, which must be set-uid `root`. It will not work, as shown below:

```
$  chmod 4777 a.out
$  su
Password:
#  chown root a.out
a.out: Not owner
```

To change the ownership, the user must either log in to the server as `root` and then make the change or move the file to a file system owned by the user's machine (for example `/usr/tmp` will usually be owned by the local machine) and make the change there.

2.2.6.2 File Operations Not Supported. Append mode and atomic writes are not guaranteed to work on remote files accessed by more than one client simultaneously. This is due to the stateless nature of the NFS protocol.

2.2.6.3 Cannot Access Remote Devices. In NFS it is not possible to access a remote mounted device or any other character or block special file or named pipes.

2.2.7 Clock Skew in User Programs

Since the NFS architecture differs in some minor ways from standard releases of the INTERACTIVE UNIX Operating System, users should be aware of those places where their own programs could run up against these incompatibilities. The section "Incompatibilities With Standard INTERACTIVE UNIX System Releases" above discusses features that do not work over the network.

Because each machine keeps its own time, the clocks may be out of sync between the NFS server and client. This might cause problems. For example, consider the following.

Many programs assume that an existing file could not have been created in the future. For example, the command `ls -l` has two basic forms of output, depending upon how old the file is:

```
server$ date
Sat Apr 12 15:27:48 GMT 1986
server$ touch file2
server$ ls -l file*
-rw-r--r--  1 jsbach          0 Dec 27  1984 file
-rw-r--r--  1 jsbach          0 Apr 12 15:27 file2
```

The first type of output from `ls` prints the year, month, and day of the last file modification if the file is more than 6 months old. The second form prints the month, day, hour, and minute of the last file modification if the file is less than 6 months old.

The `ls` command calculates the age of a file by simply subtracting the modification time of the file from the current time. If the result is greater than 6 months, the file is "old."

Assume that the time on the server is Apr 12 15:30:31, which is 3 minutes ahead of the local machine's time:

```
client$ date
Apr 12 15:27:31 GMT 1986
client$ touch file3
client$ ls -l file*
-rw-r--r--   1 jsbach          0 Dec 27  1984 file
-rw-r--r--   1 jsbach          0 Apr 12 15:27 file2
-rw-r--r--   1 jsbach          0 Apr 12  1986 file3
```

The difference between the current time and the library's modify time is a huge unsigned number, equal to −180 seconds.

Thus, `ls` believes the new file was created long ago in the past.

In general, users should remember that applications that depend upon local time and/or the file system timestamps need to deal with clock skew problems if remote files are used.

The Network Time Protocol (NTP) or "timed" programs (which are part of INTERACTIVE TCP/IP) may be used to reduce these problems.

3. RPCGEN PROTOCOL COMPILER

3.1 Introduction

The details of programming applications to use Remote Procedure Calls can be overwhelming. Perhaps most daunting is the writing of the XDR routines necessary to convert procedure arguments and results into their network format and vice versa.

Fortunately, `rpcgen` exists to help programmers write RPC applications simply and directly. The `rpcgen` command does most of the dirty work, allowing programmers to debug the main features of their application, instead of requiring them to spend most of their time debugging their network interface code.

The `rpcgen` command is a compiler. It accepts a remote program interface definition written in a language, called RPC Language, which is similar to C. It produces a C language output which includes stub versions of the client routines, a server skeleton, XDR filter routines for both parameters and results, and a header file that contains common definitions. The client stubs interface with the RPC library and effectively hide the network from their callers. The server stub similarly hides the network from the server procedures that are to be invoked by remote clients. The `rpcgen`'s output files can be compiled and linked in the usual way. The developer writes server procedures – in any language that observes C calling conventions – and links them with the server skeleton produced by `rpcgen` to get an executable server program. To use a remote program, a programmer writes an ordinary main program that makes local procedure calls to the client stubs produced by `rpcgen`. Linking this program with `rpcgen`'s stubs creates an executable program. (At present the main program must be written in C). The `rpcgen` command options can be used to suppress stub generation and to specify the transport to be used by the server stub.

Like all compilers, `rpcgen` reduces development time that would otherwise be spent coding and debugging low-level routines. All compilers, including `rpcgen`, do this at a small cost in efficiency and flexibility. However, many compilers allow escape hatches for programmers to mix low-level code with high-level code. The `rpcgen` command is no exception. In speed-critical applications, handwritten routines can be linked with the `rpcgen` output without any difficulty. Also, one may proceed by using `rpcgen` output as a starting point, and rewriting it as necessary.

3.2 Converting Local Procedures Into Remote Procedures

Assume an application that runs on a single machine, one that needs to be converted to run over the network. The following demonstrates such a conversion by way of a simple example – a program that prints a message to the console:

```
/*
 * printmsg.c: print a message on the console
 */
#include <stdio.h>

main(argc, argv)
        int argc;
        char *argv[];
{
        char *message;

        if (argc < 2) {
              fprintf(stderr, "usage: %s <message>\n", argv[0]);
              exit(1);
        }
        message = argv[1];

        if (!printmessage(message)) {
              fprintf(stderr, "%s: couldn't print your message\n",
                    argv[0]);
              exit(1);
        }
        printf("Message delivered!\n");
}
/*
 * Print a message to the console.
 * Return a boolean indicating whether the message
 * was actually printed.
 */
printmessage(msg)
        char *msg;
{
        FILE *f;

        f = fopen("/dev/console", "w");
        if (f == NULL) {
              return (0);
        }
        fprintf(f, "%s\n", msg);
        fclose(f);
        return(1);
}
```

And then, of course:

```
example% cc printmsg.c -o printmsg
example% printmsg ``Hello, there.''
Message delivered!
example%
```

If `printmessage` was turned into a remote procedure, then it could be called from anywhere in the network. Ideally, one would just like to stick a keyword like "remote" in front of a procedure to

turn it into a remote procedure. Unfortunately, one has to live within the constraints of the C language, since it existed long before RPC did. But even without language support, it is not very difficult to make a procedure remote.

In general, it is necessary to figure out what the types are for all procedure inputs and outputs. In this case, there is a procedure `printmessage` that takes a string as input and returns an integer as output. Knowing this, one can write the following protocol specification in RPC language that describes the remote version of `printmessage`:

```
/*
 * msg.x: Remote message printing protocol
 */

program MESSAGEPROG {
        version MESSAGEVERS {
                int PRINTMESSAGE(string) = 1;
        } = 1;
} = 99;
```

Since remote procedures are part of remote programs, what was actually declared was an entire remote program that contains the single procedure `PRINTMESSAGE`. This procedure was declared to be in version 1 of the remote program. No null procedure (procedure 0) is necessary because `rpcgen` generates it automatically.

Notice that everything is declared with all capital letters. This is not required, but it is a good convention to follow.

Notice also that the argument type is "string" and not "char *". This is because a "char *" in C is ambiguous. Programmers usually intend it to mean a null-terminated string of characters, but it could also represent a pointer to a single character or a pointer to an array of characters. In RPC language, a null-terminated string is unambiguously called a "string."

There are just two more things to write. First, there is the remote procedure itself. Following is the definition of a remote procedure to implement the `PRINTMESSAGE` procedure that was declared above:

```
/*
 * msg_proc.c: implementation of the remote procedure
 * "printmessage"
 */

#include <stdio.h>
#include <rpc/rpc.h>  /* always needed */
#include "msg.h"      /* need this too: msg.h will be generated */
                      /* by rpcgen */

/*
 * Remote version of "printmessage"
 */
int *
printmessage_1(msg)
        char **msg;
{
        static int result;  /* must be static! */
        FILE *f;

        f = fopen("/dev/console", "w");
        if (f == NULL) {
                result = 0;
                return (&result);
        }
        fprintf(f, "%s\n", *msg);
        fclose(f);
        result = 1;
        return (&result);
}
```

Notice that the declaration of the remote procedure `printmessage_1` differs from that of the local procedure `printmessage` in three ways:

1. It takes a pointer to a string instead of a string itself. This is true of all remote procedures; they always take pointers to their arguments rather than the arguments themselves.
2. It returns a pointer to an integer instead of an integer itself. This is also generally true of remote procedures; they always return a pointer to their results.
3. It has an "*_1*" appended to its name. In general, all remote procedures called by `rpcgen` are named by the following rule: the name in the program definition (in this example `PRINTMESSAGE`) is converted to all lower-case letters, an underbar ("_") is appended to it, and finally the version number (in this example 1) is appended.

The last thing to do is declare the main client program that will call the remote procedure:

```
/*
 * rprintmsg.c: remote version of "printmsg.c"
 */
#include <stdio.h>
#include <rpc/rpc.h>   /* always needed */
#include "msg.h"       /* need this too: msg.h will be generated */
                       /* by rpcgen */

main(argc, argv)
        int argc;
        char *argv[];
{
        CLIENT *cl;
        int *result;
        char *server;
        char *message;

        if (argc < 3) {
              fprintf(stderr, "usage: %s host message0, argv[0]);
              exit(1);
        }

        /*
         * Save values of command line arguments
         */
        server = argv[1];
        message = argv[2];

        /*
         * Create client "handle" used for calling MESSAGEPROG
         * on the server designated on the command line. We tell
         * the RPC package to use the "tcp" protocol when
         * contacting the server.
         */
        cl = clnt_create(server, MESSAGEPROG, MESSAGEVERS, "tcp");
        if (cl == NULL) {
              /*
               * Couldn't establish connection with server.
               * Print error message and die.
               */
              clnt_pcreateerror(server);
              exit(1);
        }

        /*
         * Call the remote procedure "printmessage" on the server
         */
        result = printmessage_1(&message, cl);
        if (result == NULL) {
              /*
               * An error occurred while calling the server.
               * Print error message and die.
               */
              clnt_perror(cl, server);
              exit(1);
        }

        /*
         * Okay, we successfully called the remote procedure.
         */
        if (*result == 0) {
              /*
               * Server was unable to print our message.
```

```
                * Print error message and die.
                */
               fprintf(stderr, "%s: %s couldn't print your message0,
                       argv[0], server);
               exit(1);
       }

       /*
        * The message got printed on the server's console
        */
       printf("Message delivered to %s!0, server);
}
```

There are two things to note:

1. First a client "handle" is created using the RPC library routine `clnt_create`. This client handle will be passed to the stub routines which call the remote procedure.
2. The remote procedure `printmessage_1` is called exactly the same way as it is declared in `msg_proc.c` except for the inserted client handle as the first argument.

Here is how to put all of the pieces together:

```
example% rpcgen msg.x
example% cc rprintmsg.c msg_clnt.c -lrpc -linet -o rprintmsg
example% cc msg_proc.c msg_svc.c -lrpc -linet -o msg_server
```

Two programs were just compiled: the client program `printmsg` and the server program `msg_server`. Before doing this though, `rpcgen` was used to fill in the missing pieces.

This is what `rpcgen` did with the input file `msg.x`:

1. It created a header file called `msg.h` that contained `#defines` for `MESSAGEPROG`, `MESSAGEVERS`, and `PRINTMESSAGE` for use in the other modules.
2. It created client "stub" routines in the `msg_clnt.c` file. In this case there is only one, the `printmessage_1` that was referred to from the `printmsg` client program. The name of the output file for client stub routines is always formed in this way: if the name of the input file is `FOO.x`, the client stubs output file is called `FOO_clnt.c`.
3. It created the server program which calls `printmessage_1` in `msg_proc.c`. This server program is named `msg_svc.c`. The rule for naming the server output file is similar to the previous one: for an input file called `FOO.x`, the output server file is named `FOO_svc.c`.

Try the following. First, copy the server to a remote machine and run it. For this example, the machine is called `moon`. Server processes are run in the background because they never exit.

```
moon% msg_server &
```

Then on the local machine (`sun`) you can print a message on `moon`'s console.

```
sun% printmsg moon "Hello, moon."
```

The message will get printed to `moon`'s console. You can print a message on anybody's console (including your own) with this program if you are able to copy the server to their machine and run it.

3.3 Generating XDR Routines

The previous example only demonstrated the automatic generation of client and server RPC code. The `rpcgen` command may also be used to generate XDR routines, that is, the routines necessary to convert local data structures into network format and vice versa. This example presents a complete RPC service – a remote directory listing service, which uses `rpcgen` not only to generate stub routines, but also to generate the XDR routines. Following is the protocol description file:

```
/*
 * dir.x: Remote directory listing protocol
 */
const MAXNAMELEN = 255;      /* maximum length of a directory entry */

typedef string nametype<MAXNAMELEN>;     /* a directory entry */

typedef struct namenode *namelist;       /* a link in the listing */

/*
 * A node in the directory listing
 */
struct namenode {
        nametype name;       /* name of directory entry */
        namelist next;       /* next entry */
};

/*
 * The result of a READDIR operation.
 */
union readdir_res switch (int errno) {
case 0:
        namelist list; /* no error: return directory listing */
default:
        void;        /* error occurred: nothing else to return */
};

/*
 * The directory program definition
 */
program DIRPROG {
        version DIRVERS {
                readdir_res READDIR(nametype) = 1;
        } = 1;
} = 76;
```

Running `rpcgen` on `dir.x` creates four output files. Three are the same as before: header file, client stub routines, and server skeleton. The fourth consists of the XDR routines necessary for converting the data types that were declared into XDR format and vice versa. These are output in the file `dir_xdr.c`.

Here is the implementation of the "READDIR" procedure:

```
/*
 * dir_proc.c: remote readdir implementation
 */
#include <rpc/rpc.h>
<sys/dirent.h>
#include "dir.h"

extern int errno;
extern char *malloc();
extern char *strdup();

readdir_res *
readdir_1(dirname)
        nametype *dirname;
{
        DIR *dirp;
        struct dirent *d;
        namelist nl;
        namelist *nlp;
        static readdir_res res; /* must be static! */

        /*
         * Open directory
         */
        dirp = opendir(*dirname);
        if (dirp == NULL) {
              res.errno = errno;
              return (&res);
        }

        /*
         * Free previous result
         */
        xdr_free(xdr_readdir_res, &res);

        /*
         * Collect directory entries
         */
        nlp = &res.readdir_res_u.list;
        while (d = readdir(dirp)) {
              nl = *nlp = (namenode *) malloc(sizeof(namenode));
              nl->name = strdup(d->d_name);
              nlp = &nl->next;
        }
        *nlp = NULL;

        /*
         * Return the result
         */
        res.errno = 0;
        closedir(dirp);
        return (&res);
}
```

Finally, there is the client side program to call the server:

```
/*
 * rls.c: Remote directory listing client
 */
#include <stdio.h>
#include <rpc/rpc.h>  /* always need this */
#include "dir.h"    /* need this too: will be generated by rpcgen */

extern int errno;

main(argc, argv)
        int argc;
        char *argv[];
{
        CLIENT *cl;
        char *server;
        char *dir;
        readdir_res *result;
        namelist nl;


        if (argc != 3) {
            fprintf(stderr, "usage: %s host directory\n", argv[0]);
            exit(1);
        }

        /*
         * Remember what our command line arguments refer to
         */
        server = argv[1];
        dir = argv[2];

        /*
         * Create client "handle" used for calling DIRPROG on the
         * server designated on the command line. We tell the
         * RPC package to use the "tcp" protocol
         * when contacting the server.
         */
        cl = clnt_create(server, DIRPROG, DIRVERS, "tcp");
        if (cl == NULL) {
            /*
             * Couldn't establish connection with server.
             * Print error message and die.
             */
            clnt_pcreateerror(server);
            exit(1);
        }

        /*
         * Call the remote procedure readdir on the server
         */
        result = readdir_1(&dir, cl);
        if (result == NULL) {
            /*
             * An error occurred while calling the server.
             * Print error message and die.
             */
            clnt_perror(cl, server);
            exit(1);
        }

        /*
         * Okay, we successfully called the remote procedure.
         */
```

```
        if (result->errno != 0) {
            /*
             * A remote system error occurred.
             * Print error message and die.
             */
            errno = result->errno;
            perror(dir);
            exit(1);
        }

        /*
         * Successfully got a directory listing.
         * Print it out.
         */
        for (nl = result->readdir_res_u.list; nl != NULL;
          nl = nl->next) {
            printf("%s\n", nl->name);
        }
}
```

Compile everything, and run.

```
sun% rpcgen dir.x
sun% cc rls.c dir_clnt.c dir_xdr.c -lrpc -linet -o rls
sun% cc dir_svc.c dir_proc.c dir_xdr.c -lrpc -linet -o dir_svc
sun% dir_svc &

moon% rls sun /usr/pub
.
..
ascii
eqnchar
greek
kbd
marg8
tabclr
tabs
tabs4
moon%
```

A final note about `rpcgen` is that the client program and the server procedure can be tested together as a single program by simply linking them with each other rather than with the client and server stubs. The procedure calls will be executed as ordinary local procedure calls, and the program can be debugged with a local debugger such as `sdb`. When the program is working, the client program can be linked to the client stub produced by `rpcgen`, and the server procedures can be linked to the server stub produced by `rpcgen`.

☛ Note that if you do this, you may want to comment out calls to RPC library routines and have client-side routines call server routines directly.

3.4 The C Preprocessor

Since the C preprocessor is run on all input files before they are compiled, all the preprocessor directives are legal within a `.x` file. Four symbols may be defined, depending upon which output file is getting generated:

Symbol	*Usage*
RPC_HDR	For header file output
RPC_XDR	For XDR routine output
RPC_SVC	For server skeleton output
RPC_CLNT	For client stub output

Also, `rpcgen` does a little preprocessing of its own. Any line that begins with a percent sign is passed directly into the output file, without any interpretation of the line. Here is a simple example that demonstrates the preprocessing features:

```
/*
 * time.x: Remote time protocol
 */
program TIMEPROG {
        version TIMEVERS {
                unsigned int TIMEGET(void) = 1;
        } = 1;
} = 44;

#ifdef RPC_SVC
%int *
%timeget_1()
%{
%        static int thetime;
%
%        thetime = time(0);
%        return (&thetime);
%}
#endif
```

The '%' feature is not generally recommended, as there is no guarantee that the compiler will stick the output where you intended.

3.5 RPC Language

RPC language is an extension of XDR language. The sole extension is the addition of the *program* type. For a complete description of the XDR language syntax, see section 6, "XDR PROTOCOL SPECIFICATION." For a description of the RPC extensions to the XDR language, see section 5, "RPC PROTOCOL SPECIFICATION."

However, XDR language is so close to C that if you know C, you know most of it already. Following is a description of the syntax of the RPC language, as well as a few examples. Also shown is how the various RPC and XDR type definitions get compiled into C type definitions in the output header file.

3.5.1 Definitions

An RPC language file consists of a series of definitions:

```
definition-list:
        definition ";"
        definition ";" definition-list
```

It recognizes six types of definitions:

```
definition:
        struct-definition
        union-definition
        enum-definition
        typedef-definition
        const-definition
        program-definition
```

3.5.2 Structures

An XDR structure is declared almost exactly like its C counterpart. It looks like the following:

```
struct-definition:
        "struct" struct-ident "{"
                declaration-list
        "}"

declaration-list:
        declaration ";"
        declaration ";" declaration-list
```

As an example, here is an XDR structure to define a two-dimensional coordinate, and the C structure that it gets compiled into in the output header file:

```
struct coord {          struct coord {
        int x;    -->           int x;
        int y;                  int y;
};                      };
                        typedef struct coord coord;
```

The output is identical to the input, except for the added `typedef` at the end of the output. This allows one to use "coord" instead of "struct coord" when declaring items.

3.5.3 Unions

XDR unions are discriminated unions, and they look quite different from C unions. They are more analogous to Pascal variant records than they are to C unions.

```
union-definition:
        "union" union-ident "switch" "(" declaration ")" "{"
                case-list
        "}"

case-list:
        "case" value ":" declaration ";"
        "default" ":" declaration ";"
        "case" value ":" declaration ";" case-list
```

Here is an example of a type that might be returned as the result of a "read data" operation. If there is no error, return a block of data. Otherwise, do not return anything.

```
union read_result switch (int errno) {
case 0:
        opaque data[1024];
default:
        void;
};
```

It gets compiled into the following:

```
struct read_result {
        int errno;
        union {
                char data[1024];
        } read_result_u;
};
typedef struct read_result read_result;
```

Notice that the union component of the output struct has the same name as the type name, except for the trailing "_u."

3.5.4 Enumerations

XDR enumerations have the same syntax as C enumerations:

```
enum-definition:
        "enum" enum-ident "{"
                enum-value-list
        "}"

enum-value-list:
        enum-value
        enum-value "," enum-value-list

enum-value:
        enum-value-ident
        enum-value-ident "=" value
```

Following is a short example of an XDR enum and the C enum that it gets compiled into:

```
enum colortype {                    enum colortype {
        RED = 0,                            RED = 0,
        GREEN = 1,      -->                 GREEN = 1,
        BLUE = 2                            BLUE = 2
};                                  };
                                    typedef enum colortype colortype;
```

3.5.5 Typedef

XDR typedefs have the same syntax as C typedefs:

```
typedef-definition:
        "typedef" declaration
```

Here is an example that defines a `fname_type` used for declaring file name strings that have a maximum length of 255 characters:

```
typedef string fname_type<255>;     -->     typedef char *fname_type;
```

3.5.6 Constants

XDR constants symbolize constants that may be used wherever an integer constant is used, for example, in array size specifications:

```
const-definition:
        "const" const-ident "=" integer
```

For example, the following defines a constant DOZEN equal to 12:

```
const DOZEN = 12;     -->     #define DOZEN 12
```

3.5.7 Programs

RPC programs are declared using the following syntax:

```
program-definition:
        "program" program-ident "{"
                version-list
        "}" "=" value

version-list:
        version ";"
        version ";" version-list

version:
        "version" version-ident "{"
                procedure-list
        "}" "=" value

procedure-list: .
        procedure ";"
        procedure ";" procedure-list

procedure:
        type-ident procedure-ident "(" type-ident ")" "=" value
```

For example, here is the time protocol:

```
/*
 * time.x: Get or set the time. Time is represented as
 * number of seconds since 0:00, January 1, 1970.
 */
program TIMEPROG {
        version TIMEVERS {
                unsigned int TIMEGET(void) = 1;
                void TIMESET(unsigned) = 2;
        } = 1;
} = 44;
```

This file compiles into `#defines` in the output header file:

```
#define TIMEPROG 44
#define TIMEVERS 1
#define TIMEGET 1
#define TIMESET 2
```

3.5.8 Declarations

In XDR, there are only four kinds of declarations:

```
declaration:
        simple-declaration
        fixed-array-declaration
        variable-array-declaration
        pointer-declaration
```

1) Simple Declarations are just like simple C declarations:

```
simple-declaration:
        type-ident variable-ident
```

Example:

```
colortype color;    -->    colortype color;
```

2) Fixed-length Array Declarations are just like C array declarations:

```
fixed-array-declaration:
        type-ident variable-ident "[" value "]"
```

Example:

```
colortype palette[8];    -->    colortype palette[8];
```

3) Variable-Length Array Declarations have no explicit syntax in C, so XDR invents its own using angle-brackets:

```
variable-array-declaration:
        type-ident variable-ident "<" value ">"
        type-ident variable-ident "<" ">"
```

The maximum size is specified between the angle brackets. The size may be omitted, indicating that the array may be of any size:

```
int heights<12>;        /* at most 12 items */
int widths<>;           /* any number of items */
```

Since variable-length arrays have no explicit syntax in C, these declarations are actually compiled into "structs." For example, the "heights" declaration gets compiled into the following struct:

```
struct {
        uint heights_len;       /* # of items in array */
        int *heights_val;       /* pointer to array */
} heights;
```

Note that the number of items in the array is stored in the `_len` component, and the pointer to the array is stored in the `_val` component. The first part of each of these component's names is the same as the name of the declared XDR variable.

4) Pointer Declarations are made in XDR exactly as they are in C. You cannot really send pointers over the network, but you can use XDR pointers for sending recursive data types such as lists and trees. The type is actually called "optional-data," not "pointer," in XDR language:

```
pointer-declaration:
        type-ident "*" variable-ident
```

Example:

```
listitem *next;     -->     listitem *next;
```

3.5.9 Special Cases

There are a few exceptions to the rules described above.

Booleans: C has no built-in boolean type. However, the RPC library does have a boolean type called `bool_t` that is either `TRUE` or `FALSE`. Things declared as type `bool` in XDR language are compiled into `bool_t` in the output header file.

Example:

```
bool married;     -->     bool_t married;
```

Strings: C has no built-in string type, but instead uses the null-terminated "char *" convention. In XDR language, strings are declared using the "string" keyword and compiled into "char *"s in the output header file. The maximum size contained in the angle brackets specifies the maximum number of characters allowed in the strings (not counting the `NULL` character). The maximum size may be left off, indicating a string of arbitrary length.

Examples:

```
string name<32>;        -->    char *name;
string longname<>;      -->    char *longname;
```

Opaque Data: Opaque data is used in RPC and XDR to describe untyped data, that is, just sequences of arbitrary bytes. It may be declared either as a fixed or variable length array.
Examples:

```
opaque diskblock[512];      -->    char diskblock[512];

opaque filedata<1024>;      -->    struct {
                                       uint filedata_len;
                                       char *filedata_val;
                                   } filedata;
```

Voids: In a void declaration, the variable is not named. The declaration is just "void" and nothing else. Void declarations can only occur in two places: union definitions and program definitions (as the argument or result of a remote procedure).

4. RPC GUIDE

4.1 Introduction

This section is intended for programmers who wish to write network applications using remote procedure calls (explained below), thus avoiding low-level system primitives based on sockets. The programmer must be familiar with the C programming language and should have a working knowledge of network theory.

Programs that communicate over a network need a paradigm for communication. A low-level mechanism might send a signal on the arrival of incoming packets, causing a network signal handler to execute. A high-level mechanism would be the Ada* `rendezvous`. The method used by the NFS is the Remote Procedure Call (RPC) paradigm, in which a client communicates with a server. In this process, the client first calls a procedure to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs the service requested, sends back the reply, and the procedure call returns to the client.

The RPC interface is divided into three layers. The highest layer is totally transparent to the programmer. To illustrate, at this level a program can contain a call to `rnusers()`, which returns the number of users on a remote machine. The user need not be aware that RPC is being used, since the call is simply made in a program, just as `malloc()` would be called.

At the middle layer, the routines `registerrpc()` and `callrpc()` are used to make RPC calls: `registerrpc()` obtains a unique system-wide number, while `callrpc()` executes a remote procedure call. The `rnusers()` call is implemented using these two routines. The middle-layer routines are designed for most common applications and shield the user from needing to know about sockets.

The lowest layer is used for more sophisticated applications, which may want to alter the defaults of the routines. At this layer, sockets used for transmitting RPC messages can be explicitly manipulated. This level should be avoided if possible.

Although this section only discusses the interface to C, remote procedure calls can be made from any language. Even though this section discusses RPC when it is used to communicate between processes on different machines, it works just as well for communication between different processes on the same machine.

Following is a diagram of the RPC paradigm:

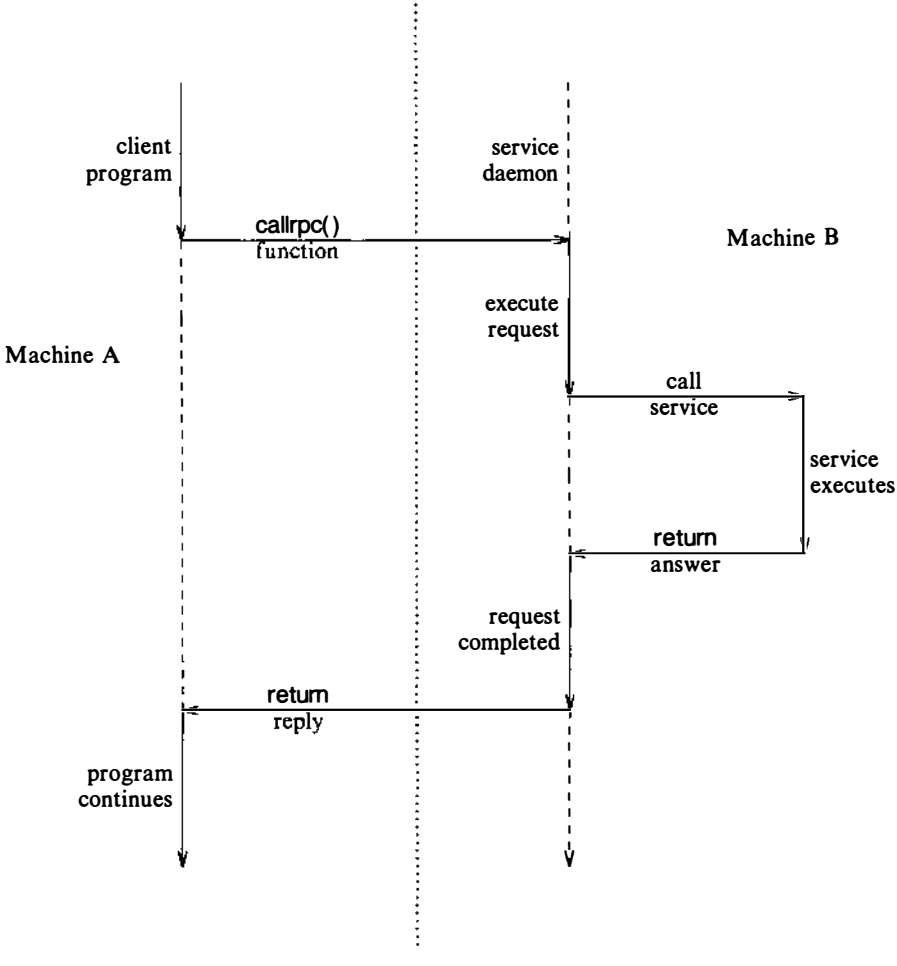


Figure 4. Network Communications With RPC

4.2 Introductory Examples

4.2.1 Highest Layer

Consider a program that needs to know how many users are logged in to a remote machine. This can be done by calling the library routine `rnusers()` as illustrated below:

```
#include <stdio.h>

main(argc, argv)
        int argc;
        char **argv;
{
        unsigned num;

        if (argc < 2) {
                fprintf(stderr, "usage: rnusers hostname\n");
                exit(1);
        }
        if ((num = rnusers(argv[1])) < 0) {
                fprintf(stderr, "error: rnusers\n");
                exit(-1);
        }
        printf("%d users on %s\n", num, argv[1]);
        exit(0);
}
```

RPC library routines such as `rnusers()` are included in the C library `librpcsvc.a`. Thus, the program above could be compiled with:

```
$ cc program.c -lrpcsvc -lrpc -linet
```

Some other library routines are `rstat()` to gather remote performance statistics and `ypmatch()` to glean information from the NIS. The NIS library routines are documented in the manual entry *ypclnt*(3N) in the *INTERACTIVE Network Information Service Guide*.

4.2.2 Intermediate Layer

The simplest interface, which explicitly makes RPC calls, uses the functions `callrpc()` and `registerrpc()`. Using this method, another way to get the number of remote users is:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

main(argc, argv)
        int argc;
        char **argv;
{
        unsigned long nusers;

        if (argc < 2) {
                fprintf(stderr, "usage: nusers hostname\n");
                exit(-1);
        }
        if (callrpc(argv[1], RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM,
            xdr_void, 0, xdr_u_long, &nusers) != 0) {
                fprintf(stderr, "error: callrpc\n");
                exit(1);
        }
        printf("number of users on %s is %d\n", argv[1], nusers);
        exit(0);
}
```

A program number, version number, and procedure number define each RPC procedure. The program number defines a group of related remote procedures, each of which has a different procedure number. Each program also has a version number, so when a minor change is made to a remote service (adding a new procedure, for example), a new program number does not have to be assigned.

When a procedure is to be called to find the number of remote users, the appropriate program, version, and procedure numbers are looked up in a manual, in a manner similar to looking up the name of memory allocator when memory is to be allocated.

The simplest routine in the RPC library used to make remote procedure calls is `callrpc()`. It has eight parameters. The first is the name of the remote machine. The next three parameters are the program, version, and procedure numbers. The following two parameters define the argument of the RPC call, and the final two parameters are for the return value of the call. If it completes successfully, `callrpc()` returns zero, but nonzero otherwise. The exact meaning of the return codes is found in `<rpc/clnt.h>` and is, in fact, an `enum clnt_stat` cast into an integer.

Since data types may be represented differently on different machines, `callrpc()` needs both the type of the RPC argument and a pointer to the argument itself (and similarly for the result). For `RUSERSPROC_NUM`, since the return value is an `unsigned long`, `callrpc()` has `xdr_u_long` as its first return parameter, which says that the result is of type

`unsigned long`, and `&nusers` as its second return parameter, which is a pointer to where the `long` result will be placed. Since `RUSERSPROC_NUM` takes no argument, the argument parameter of `callrpc()` is `xdr_void`.

After trying several times to deliver a message, if `callrpc()` gets no answer, it returns with an error code. The delivery mechanism is UDP, which stands for User Datagram Protocol. Methods for adjusting the number of retries or for using a different protocol require the use of the lower layer of the RPC library, discussed later in this section. The remote server procedure corresponding to the above might look like this:

```
char *
nuser(indata)
        char *indata;
{
        static int nusers;

        /*
         * code here to compute the number of users
         * and place result in variable nusers
         */
        return ((char *)&nusers);
}
```

It takes one argument, which is a pointer to the input of the remote procedure call (ignored in the above example), and it returns a pointer to the result. In the current version of C, since character pointers are the generic pointers, both the input argument and the return value are cast to `char (*`.

Normally, a server registers all of the RPC calls it plans to handle and then goes into an infinite loop waiting to service requests. In this example, there is only one procedure to register, so the main body of the server would look like this:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

char *nuser();

main()
{
        registerrpc(RUSERSPROG, RUSERSVERS, RUSERSPROC_NUM, nuser,
            xdr_void, xdr_u_long);
        svc_run();                  /* never returns */
        fprintf(stderr, "Error: svc_run returned!\n");
        exit(1);
}
```

The `registerrpc()` routine establishes what C procedure corresponds to each RPC procedure number. The first three

parameters, `RUSERPROG`, `RUSERSVERS`, and `RUSERSPROC_NUM`, are the program, version, and procedure numbers of the remote procedure to be registered; `nuser` is the name of the C procedure implementing it; and `xdr_void` and `xdr_u_long` are the types of the input to and output from the procedure.

Only the UDP transport mechanism can use `registerrpc()`; thus, it is always safe in conjunction with calls generated by `callrpc()`.

☛ The UDP transport mechanism can only deal with arguments and results that are reasonably small. The limit is typically 4 to 8 KB in length.

4.2.3 Assigning Program Numbers

Program numbers are assigned in groups of 0x20000000 (536870912) according to the following chart:

0	-	1fffffff	defined by Sun Microsystems
20000000	-	3fffffff	defined by user
40000000	-	5fffffff	transient
60000000	-	7fffffff	reserved
80000000	-	9fffffff	reserved
a0000000	-	bfffffff	reserved
c0000000	-	dfffffff	reserved
e0000000	-	ffffffff	reserved

Sun Microsystems* administers the first group of numbers, and the intent is that they will be identical across all systems and applications. If a customer develops an application that might be of general interest, that application should be given a number assigned by Sun* from the first range. The second group of numbers is reserved for specific customer applications; this range is intended primarily for debugging new programs. The third group is reserved for applications that generate program numbers dynamically. The final groups are reserved for future use and should not be used.

4.2.4 Passing Arbitrary Data Types

In the previous example, the RPC call passes a single `unsigned long`. RPC can handle arbitrary data structures, regardless of different machines' byte orders or structure layout conventions by always converting them to a network standard called

eXternal Data Representation (XDR) before sending them over the wire. The process of converting from a particular machine representation to XDR format is called *serializing*, and the reverse process is called *deserializing*. The type field parameters of `callrpc()` and `registerrpc()` can be a built-in procedure like `xdr_u_long()` in the previous example, or a user supplied one. XDR has these built-in type routines:

```
xdr_int()      xdr_u_int()     xdr_enum()
xdr_long()     xdr_u_long()    xdr_bool()
xdr_short()    xdr_u_short()   xdr_string()
```

As an example of a user-defined type routine, if it was wished to send the structure:

```
struct simple {
        int a;
        short b;
} simple;
```

then `callrpc` should be called as:

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_simple, &simple ...);
```

where `xdr_simple()` is written as:

```
#include <rpc/rpc.h>

xdr_simple(xdrsp, simplep)
        XDR *xdrsp;
        struct simple *simplep;
{
        if (!xdr_int(xdrsp, &simplep->a))
                return (0);
        if (!xdr_short(xdrsp, &simplep->b))
                return (0);
        return (1);
}
```

An XDR routine returns nonzero (true in the sense of C) if it completes successfully, and zero otherwise. Since a complete description of XDR is in section 6, "XDR PROTOCOL SPECIFICATION," this section only gives a few examples of XDR implementation.

In addition to the built-in primitives, there are also the prefabricated building blocks:

```
xdr_array()          xdr_bytes()
xdr_reference()      xdr_union()
```

To send a variable array of integers, they might be packaged up as a structure like this:

```
struct varintarr {
        int *data;
        int arrlnth;
} arr;
```

and make an RPC call such as:

```
callrpc(hostname, PROGNUM, VERSNUM, PROCNUM, xdr_varintarr, &arr ...);
```

with `xdr_varintarr()` defined as:

```
xdr_varintarr(xdrsp, arrp)
     XDR *xdrsp;
     struct varintarr *arrp;
{
     return (xdr_array(xdrsp, &arrp->data, &arrp->arrlnth, MAXLEN,
         sizeof(int), xdr_int));
}
```

This routine takes as parameters the XDR handle, a pointer to the array, a pointer to the size of the array, the maximum allowable array size, the size of each array element, and an XDR routine for handling each array element.

If the size of the array is known in advance, then the following could also be used to send out an array of length `SIZE`:

```
int intarr[SIZE];

xdr_intarr(xdrsp, intarr)
        XDR *xdrsp;
        int intarr[];
{
        int i;

        for (i = 0; i < SIZE; i++) {
                if (!xdr_int(xdrsp, &intarr[i]))
                        return (0);
        }
        return (1);
}
```

XDR always converts quantities to 4-byte multiples when deserializing. Thus, if either of the examples above involved characters instead of integers, each character would occupy 32 bits. That is the reason for the XDR routine `xdr_bytes()`, which is like `xdr_array()` except that it packs characters. It has four parameters which are the same as the first four parameters of `xdr_array()`. For null-terminated strings, there is also the `xdr_string()` routine, which is the same as `xdr_bytes()` without the length parameter. On serializing it gets the string length from `strlen()`, and on deserializing it creates a null-terminated string.

Here is a final example that calls the previously written `xdr_simple()` as well as the built-in functions `xdr_string()` and `xdr_reference()`, which chases pointers:

```
struct finalexample {
        char *string;
        struct simple *simplep;
} finalexample;

xdr_finalexample(xdrsp, finalp)
        XDR *xdrsp;
        struct finalexample *finalp;
{
        if (!xdr_string(xdrsp, &finalp->string, MAXSTRLEN))
                return (0);
        if (!xdr_reference(xdrsp, &finalp->simplep,
            sizeof(struct simple), xdr_simple))
                return (0);
        return (1);
}
```

4.3 Lower Layers of RPC

In the examples given so far, RPC takes care of many details automatically. This section shows how to change the defaults by using lower layers of the RPC library. It is assumed that the reader is familiar with sockets and the system calls for dealing with them.

4.3.1 More on the Server Side

A number of assumptions are built into `registerrpc()`. One is that the UDP datagram protocol is being used. Another is that the user does not want to do anything unusual while deserializing, since the deserialization process happens automatically before the user's server routine is called. The server for the `nusers` program shown below is written using a lower layer of the RPC package, which does not make these assumptions:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>

int nuser();

main()
{
        SVCXPRT *transp;

        transp = svcudp_create(RPC_ANYSOCK);
        if (transp == NULL) {
              fprintf(stderr, "couldn't create an RPC server\n");
              exit(1);
        }
        pmap_unset(RUSERSPROG, RUSERSVERS);
        if (!svc_register(transp, RUSERSPROG, RUSERSVERS, nuser,
            IPPROTO_UDP)) {
              fprintf(stderr, "couldn't register RUSER service\n");
              exit(1);
        }
        svc_run();  /* never returns */
        fprintf(stderr, "should never reach this point\n");
}

nuser(rqstp, transp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        unsigned long nusers;

        switch (rqstp->rq_proc) {
        case NULLPROC:
             if (!svc_sendreply(transp, xdr_void, 0)) {
                    fprintf(stderr, "couldn't reply to RPC call\n");
                    exit(1);
                }
                return;
        case RUSERSPROC_NUM:
                /*
                 * code here to compute the number of users
                 * and put in variable nusers
                 */
             if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
                    fprintf(stderr, "couldn't reply to RPC call\n");
                    exit(1);
                }
                return;
        default:
                svcerr_noproc(transp);
                return;
        }
}
```

First, the server gets a transport handle, which is used for sending out RPC messages. The `registerrpc()` command uses `svcudp_create()` to get a UDP handle. If a reliable protocol is required, `svctcp_create()` should be called instead. If the argument to `svcudp_create()` is `RPC_ANYSOCK`, the RPC library creates a socket on which to send out RPC calls. Otherwise,

`svcudp_create()` expects its argument to be a valid socket number. If the user specifies his own socket, it can be bound or unbound. If it is bound to a port by the user, the port numbers of `svcudp_create()` and `clntudp_create()` (the low-level client routine) must match.

When the user specifies `RPC_ANYSOCK` for a socket or gives an unbound socket, the system determines port numbers in the following way: when a server starts up, it advertises to a port mapper daemon on its local machine, which picks a port number for the RPC procedure if the socket specified to `svcudp_create()` is not already bound. When the `clntudp_create()` call is made with an unbound socket, the system queries the port mapper on the machine to which the call is being made and gets the appropriate port number. If the port mapper is not running or has no port corresponding to the RPC call, the RPC call fails. Users can make RPC calls to the port mapper themselves. The appropriate procedure numbers are contained in the `include` file `<rpc/pmap_prot.h>`.

After creating an `SVCXPRT`, the next step is to call `pmap_unset()` so that if the `nusers` server crashed earlier, any previous trace of it is erased before restarting. More precisely, `pmap_unset()` erases the entry for `RUSERS` from the port mapper's tables.

Finally, the program number for `nusers` is associated with the procedure `nuser()`. The final argument to `svc_register()` is normally the protocol being used which, in this case, is `IPPROTO_UDP`. Notice that unlike `registerrpc()`, there are no XDR routines involved in the registration process. Also, registration is done on the program, rather than procedure, level.

The user routine `nuser()` must call and dispatch the appropriate XDR routines based on the procedure number. Note that two things are handled by `nuser()` which are handled automatically by `registerrpc()`. The first is that procedure `NULLPROC` (currently zero) returns with no arguments. This can be used as a simple test for detecting if a remote program is running. Second, there is a check for invalid procedure numbers. If one is detected, `svcerr_noproc()` is called to handle the error.

The user service routine serializes the results and returns them to the RPC caller via `svc_sendreply()`. Its first parameter is the `SVCXPRT` handle, the second is the XDR routine, and the third

is a pointer to the data to be returned. Not illustrated above is how a server handles an RPC program that passes data. As an example, a procedure, `RUSERSPROC_BOOL`, which has an argument `nusers` and returns `TRUE` or `FALSE` depending on whether there are `nusers` logged on can be added. It would look like this:

```
case RUSERSPROC_BOOL: {
        int bool;
        unsigned nuserquery;

        if (!svc_getargs(transp, xdr_u_int, &nuserquery)) {
                svcerr_decode(transp);
                return;
        }
        /*
         * code to set nusers = number of users
         */
        if (nuserquery == nusers)
                bool = TRUE;
        else
                bool = FALSE;
        if (!svc_sendreply(transp, xdr_bool, &bool)) {
                fprintf(stderr, "couldn't reply to RPC call\n");
                exit(1);
        }
        return;
}
```

The relevant routine is `svc_getargs()`, which takes as arguments an `SVCXPRT` handle, the XDR routine, and a pointer to where the input is to be placed.

4.3.2 Memory Allocation With XDR

XDR routines not only do input and output, they also do memory allocation. This is why the second parameter of `xdr_array()` is a pointer to an array, rather than the array itself. If it is `NULL`, then `xdr_array()` allocates space for the array and returns a pointer to it, putting the size of the array in the third argument. As an example, consider the following XDR routine, `xdr_chararr1()`, which deals with a fixed array of bytes with length `SIZE`:

```
xdr_chararr1(xdrsp, chararr)
        XDR *xdrsp;
        char chararr[];
{
        char *p;
        int len;

        p = chararr;
        len = SIZE;
        return (xdr_bytes(xdrsp, &p, &len, SIZE));
}
```

It might be called from a server like this,

```
char chararr[SIZE];

svc_getargs(transp, xdr_chararr1, chararr);
```

where `chararr` has already allocated space. If XDR was wanted to do the allocation, this routine would have to be rewritten in the following way:

```
xdr_chararr2(xdrsp, chararrp)
        XDR *xdrsp;
        char **chararrp;
{
        int len;

        len = SIZE;
        return (xdr_bytes(xdrsp, chararrp, &len, SIZE));
}
```

The RPC call might then look like this:

```
char *arrptr;

arrptr = NULL;
svc_getargs(transp, xdr_chararr2, &arrptr);
/*
 * use the result here
 */
svc_freeargs(xdrsp, xdr_chararr2, &arrptr);
```

After using the character array, it can be freed with `svc_freeargs()`. In the routine `xdr_finalexample()` given earlier, if `finalp->string` was `NULL` in the call

```
svc_getargs(transp, xdr_finalexample, &finalp);
```

then

```
svc_freeargs(xdrsp, xdr_finalexample, &finalp);
```

frees the array allocated to hold `finalp->string`; otherwise, it frees nothing. The same is true for `finalp->simplep`.

To summarize, each XDR routine is responsible for serializing, deserializing, and allocating memory. When an XDR routine is called from `callrpc()`, the serializing part is used. When called from `svc_getargs()`, the deserializer is used. When called from `svc_freeargs()`, the memory deallocator is used. When building simple examples like those in this section, a user does not have to worry about the three modes.

4.3.3 The Calling Side

When `callrpc` is used, there is no control over the RPC delivery mechanism or the socket used to transport the data. To illustrate

the layer of RPC that allows adjustment of these parameters, consider the following code to call the `nusers` service:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/rusers.h>
#include <sys/socket.h>
#include <sys/fs/nfs/time.h>
#include <netdb.h>

main(argc, argv)
     int argc;
     char **argv;
{
     struct hostent *hp;
     struct timeval pertry_timeout, total_timeout;
     struct sockaddr_in server_addr;
     int addrlen, sock = RPC_ANYSOCK;
     register CLIENT *client;
     enum clnt_stat clnt_stat;
     unsigned long nusers;

     if (argc < 2) {
             fprintf(stderr, "usage: nusers hostname\n");
             exit(-1);
     }
     if ((hp = gethostbyname(argv[1])) == NULL) {
             fprintf(stderr, "cannot get addr for '%s'\n", argv[1]);
             exit(-1);
     }

     pertry_timeout.tv_sec = 3;
     pertry_timeout.tv_usec = 0;
     addrlen = sizeof(struct sockaddr_in);
     bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, hp->h_length);
     server_addr.sin_family = AF_INET;
     server_addr.sin_port =  0;
     if ((client = clntudp_create(&server_addr, RUSERSPROG,
         RUSERSVERS, pertry_timeout, &sock)) == NULL) {
             perror("clntudp_create");
             exit(-1);
     }
     total_timeout.tv_sec = 20;
     total_timeout.tv_usec = 0;
     clnt_stat = clnt_call(client, RUSERSPROC_NUM, xdr_void, 0,
         xdr_u_long, &nusers, total_timeout);
     if (clnt_stat != RPC_SUCCESS) {
             clnt_perror(client, "rpc");
             exit(-1);
     }
     clnt_destroy(client);
}
```

The low-level version of `callrpc()` is `clnt_call()`. It takes a `CLIENT` pointer rather than a host name. The parameters to `clnt_call()` are a `CLIENT` pointer, the procedure number, the XDR routine for serializing the argument, a pointer to the argument, the XDR routine for deserializing the return value, a pointer to where the return value will be placed, and the time in seconds to wait for a reply.

The `CLIENT` pointer is encoded with the transport mechanism. The `callrpc()` command uses UDP; thus it calls `clntudp_create()` to get a `CLIENT` pointer. To get TCP (Transmission Control Protocol), `clnttcp_create()` would be used.

The parameters to `clntudp_create()` are the server address, the length of the server address, the program number, the version number, a timeout value (between tries), and a pointer to a socket. The final argument to `clnt_call()` is the total time to wait for a response. Thus, the number of tries is the `clnt_call()` timeout divided by the `clntudp_create()` timeout.

One thing should be noted when using the `clnt_destroy()` call – it deallocates any space associated with the `CLIENT` handle, but it does not close the socket associated with it, which was passed as an argument to `clntudp_create()`. The reason is that if there are multiple client handles using the same socket, then it is possible to close one handle without destroying the socket that other handles are using.

To make a stream connection, the call to `clntudp_create()` is replaced with a call to `clnttcp_create()`:

```
clnttcp_create(&server_addr, prognum, versnum, &socket, inputsize,
        outputsize);
```

There is no timeout argument; instead the receive and send buffer sizes must be specified. When the `clnttcp_create()` call is made, a TCP connection is established. All RPC calls using that `CLIENT` handle would use this connection. The server side of an RPC call using TCP has `svcudp_create()` replaced by `svctcp_create()`.

4.4 Other RPC Features

This section discusses some other aspects of RPC.

4.4.1 Select on the Server Side

Suppose a process is processing RPC requests while performing some other activity. If the other activity involves periodically updating a data structure, the process can set an alarm signal before calling `svc_run()`. However, if the other activity involves waiting for a file descriptor, the `svc_run()` call will not work. The code for `svc_run()` is as follows:

```
void
svc_run()
{
        int readfds;

        for (;;) {
                readfds = svc_fds;
                switch (select(32, &readfds, NULL, NULL, NULL)) {

                case -1:
                        if (errno == EINTR)
                                continue;
                        perror("rstat: select");
                        return;
                case 0:
                        break;
                default:
                        svc_getreq(readfds);
                }
        }
}
```

The `svc_run()` command can be bypassed, and `svc_getreq()` called directly. To do this the file descriptors of the socket(s) associated with the programs which are being waited for must be known. Thus, users can write their own `selects`, which wait on both the RPC socket and their own descriptors.

4.4.2 Broadcast RPC

The `pmap` and RPC protocols implement broadcast RPC. Here are the main differences between broadcast RPC and normal RPC calls:

1. Normal RPC expects one answer, whereas broadcast RPC expects many answers (one or more answer from each responding machine).
2. Broadcast RPC can only be supported by packet-oriented (connectionless) transport protocols like UDP/IP.
3. The implementation of broadcast RPC treats all unsuccessful responses as garbage by filtering them out. Thus, if there is a version mismatch between the broadcaster and a remote service, the user of broadcast RPC never knows.
4. All broadcast messages are sent to the portmap port. Thus, only services that register themselves with their portmapper are accessible via the broadcast RPC mechanism.

4.4.2.1 Broadcast RPC Synopsis.

```
#include <rpc/pmap_clnt.h>

enum clnt_stat clnt_stat;

clnt_stat =
clnt_broadcast(prog, vers, proc, xargs, argsp, xresults, resultsp,
eachresult)
ulong           prog;           /* program number */
ulong           vers;           /* version number */
ulong           proc;           /* procedure number */
xdrproc_t       xargs;          /* xdr routine for args */
caddr_t         argsp;          /* pointer to args */
xdrproc_t       xresults;       /* xdr routine for results */
caddr_t         resultsp;       /* pointer to results */
bool_t          (*eachresult)(); /* call with each result */
                                /* obtained */
```

The procedure `eachresult()` is called each time a valid result is obtained. It returns a boolean that indicates whether or not the client wants more responses:

```
bool_t done;

done = eachresult(resultsp, raddr)
caddr_t                 resultsp;
struct sockaddr_in      *raddr; /* address of machine that sent */
                                /* response */
```

If `done` is `TRUE`, broadcasting stops and `clnt_broadcast()` returns successfully. Otherwise, the routine waits for another response. The request is rebroadcast after a few seconds of waiting. If no responses come back, the routine returns with `RPC_TIMEDOUT`. To interpret `clnt_stat` errors, feed the error code to `clnt_perrno()`.

4.4.3 Batching

The RPC architecture is designed so that clients send a call message and wait for servers to reply that the call succeeded. This implies that clients do not compute while servers are processing a call. This is inefficient if the client does not want or need an acknowledgment for every message sent. It is possible for clients to continue computing while waiting for a response, using RPC batch facilities.

RPC messages can be placed in a "pipeline" of calls to a desired server; this is called *batching*. Batching assumes that:

1. Each RPC call in the pipeline requires no response from the server, and the server does not send a response message.

2. The pipeline of calls is transported on a reliable byte stream transport such as TCP/IP (Transmission Control Protocol/Internet Protocol).

Since the server does not respond to every call, the client can generate new calls in parallel with the server executing previous calls. Furthermore, the TCP/IP implementation can buffer up many call messages and send them to the server in one `write` system call. This overlapped execution greatly decreases the interprocess communication overhead of the client and server processes and the total elapsed time of a series of calls.

Since the batched calls are buffered, the client should eventually do a legitimate call in order to flush the pipeline.

A contrived example of batching follows. Assume a string rendering service (like a window system) has two similar calls – one renders a string and returns void results, while the other renders a string and remains silent. The service (using the TCP/IP transport) may look like:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>

void windowdispatch();

main()
{
    SVCXPRT *transp;

    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
            fprintf(stderr, "couldn't create an RPC server\n");
            exit(1);
    }
    pmap_unset(WINDOWPROG, WINDOWVERS);
    if (!svc_register(transp, WINDOWPROG, WINDOWVERS, windowdispatch,
        IPPROTO_TCP)) {
            fprintf(stderr, "couldn't register WINDOW service\n");
            exit(1);
    }
    svc_run();  /* never returns */
    fprintf(stderr, "should never reach this point\n");
}
```

```
void
windowdispatch(rqstp, transp)
        struct svc_req *rqstp;
        SVCXPRT *transp;
{
        char *s = NULL;

        switch (rqstp->rq_proc) {
        case NULLPROC:
              if (!svc_sendreply(transp, xdr_void, 0)) {
                    fprintf(stderr, "couldn't reply to RPC call\n");
                    exit(1);
              }
              return;
        case RENDERSTRING:
              if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                    fprintf(stderr, "couldn't decode arguments\n");
                    svcerr_decode(transp); /* tell caller he */
                                           /* screwed up */
                    break;
              }
              /*
               * call here to to render the string s
               */
              if (!svc_sendreply(transp, xdr_void, NULL)) {
                    fprintf(stderr, "couldn't reply to RPC call\n");
                    exit(1);
              }
              break;

        case RENDERSTRING_BATCHED:
              if (!svc_getargs(transp, xdr_wrapstring, &s)) {
                    fprintf(stderr, "couldn't decode arguments\n");
                    /*
                     * we are silent in the face of protocol
                     * errors
                     */
                    break;
              }
              /*
               * call here to to render the string s,
               * but sends no reply!
               */
              break;
        default:
              svcerr_noproc(transp);
              return;
        }
        /*
         * now free string allocated while decoding arguments
         */
        svc_freeargs(transp, xdr_wrapstring, &s);
}
```

Of course the service could have one procedure that takes the string and a boolean to indicate whether or not the procedure should respond.

In order for a client to take advantage of batching, the client must perform RPC calls on a TCP/IP-based transport, and the actual calls must have the following attributes:

1. The result's XDR routine must be 0 (NULL).
2. The RPC call's timeout must be 0.

Following is an example of a client that uses batching to render a bunch of strings; the batching is flushed when the client gets a null string:

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <rpcsvc/windows.h>
#include <sys/socket.h>
#include <sys/fs/nfs/time.h>
#include <netdb.h>

main(argc, argv)
        int argc;
        char **argv;
{
        struct hostent *hp;
        struct timeval pertry_timeout, total_timeout;
        struct sockaddr_in server_addr;
        int addrlen, sock = RPC_ANYSOCK;
        register CLIENT *client;
        enum clnt_stat clnt_stat;
        char buf[1000];
        char *s = buf;


        /*
         * initial as in example 3.3
         */
        if ((client = clnttcp_create(&server_addr, WINDOWPROG,
            WINDOWVERS, &sock, 0, 0)) == NULL) {
                perror("clnttcp_create");
                exit(-1);
        }
        total_timeout.tv_sec = 0;
        total_timeout.tv_usec = 0;
        while (scanf("%s", s) != EOF) {
                clnt_stat = clnt_call(client, RENDERSTRING_BATCHED,
                    xdr_wrapstring, &s, NULL, NULL, total_timeout);
                if (clnt_stat != RPC_SUCCESS) {
                        clnt_perror(client, "batched rpc");
                        exit(-1);
                }
        }

        /*
         * now flush the pipeline
         */
        total_timeout.tv_sec = 20;
        clnt_stat = clnt_call(client, NULLPROC,
            xdr_void, NULL, xdr_void, NULL, total_timeout);
        if (clnt_stat != RPC_SUCCESS) {
                clnt_perror(client, "rpc");
                exit(-1);
        }

        clnt_destroy(client);
}
```

Since the server sends no message, the clients cannot be notified of any of the failures that may occur. Therefore, clients are on their own when it comes to handling errors.

The above example was completed to render all of the (2000) lines in the file `/etc/termcap`. The rendering service did nothing but to throw the lines away. The example was run in the following four configurations:

1. Machine to itself, regular RPC
2. Machine to itself, batched RPC
3. Machine to another, regular RPC
4. Machine to another, batched RPC

The results are as follows:

1. 50 seconds
2. 16 seconds
3. 52 seconds
4. 10 seconds

Running `fscanf()` on `/etc/termcap` only requires 6 seconds. These timings show the advantage of protocols that allow for overlapped execution, though these protocols are often hard to design.

4.4.4 Authentication

In the examples presented so far, the caller never identified itself to the server, and the server never required an ID from the caller. Clearly, some network services, such as a network file system, require stronger security measures than those that have been presented so far.

In reality, every RPC call is authenticated by the RPC package on the server, and similarly, the RPC client package generates and sends authentication parameters. Just as different transports (TCP/IP or UDP/IP) can be used when creating RPC clients and servers, different forms of authentication can be associated with RPC clients; the default authentication type used as a default is type `none`.

The authentication subsystem of the RPC package is open ended; that is, numerous types of authentication are easy to support.

However, this section deals only with INTERACTIVE UNIX Operating System type authentication, which besides `none` is the only supported type.

4.4.4.1 The Client Side. When a caller creates a new RPC client handle as in:

```
clnt = clntudp_create(address, prognum, versnum, wait, sockp)
```

the appropriate transport instance defaults the associate authentication handle to be:

```
clnt->cl_auth = authnone_create();
```

The RPC client can choose to use the INTERACTIVE UNIX Operating System type authentication by setting `clnt->cl_auth` after creating the RPC client handle:

```
clnt->cl_auth = authunix_create_default();
```

This causes each RPC call associated with `clnt` to carry with it the following authentication credentials structure:

```
/*
 * UNIX type credentials.
 */
struct authunix_parms {
      ulong    aup_time;        /* credentials creation time */
      char     *aup_machname;   /* host name of client machine */
      int      aup_uid;         /* client's UNIX effective uid */
      int      aup_gid;         /* client's current UNIX group ID */
      uint     aup_len;         /* the element length of aup_gids */
                                /* array */
      int      *aup_gids;       /* array of groups to which user */
                                /* belongs */
};
```

These fields are set by `authunix_create_default()` by invoking the appropriate system calls.

Since the RPC user created this new type of authentication, he is responsible for destroying it with:

```
auth_destroy(clnt->cl_auth);
```

4.4.4.2 The Server Side. It is more difficult for service implementors dealing with authentication issues since the RPC package passes the service dispatch routine a request that has an arbitrary authentication type associated with it. Consider the fields of a request handle passed to a service dispatch routine:

```
/*
 * An RPC Service request
 */
struct svc_req {
      ulong          rq_prog;       /* service program number */
      ulong          rq_vers;       /* service protocol version */
                                    /* number */
      ulong          rq_proc;       /* the desired procedure number*/
      struct opaque_auth rq_cred;   /* raw credentials from the */
                                    /* "wire" */
      caddr_t        rq_clntcred;   /* read only, cooked credentials */
};
```

The `rq_cred` is mostly opaque, except for one field of interest – the type of authentication credentials:

```
/*
 * Authentication info.  Mostly opaque to the programmer.
 */
struct opaque_auth {
        enum_t  oa_flavor;      /* type of credentials */
        caddr_t oa_base;        /* address of more auth stuff */
        uint    oa_length;      /* not to exceed MAX_AUTH_BYTES */
};
```

The RPC package guarantees the following to the service dispatch routine:

1. That the request's `rq_cred` is well formed. Thus, the service implementor may inspect the request's `rq_cred.oa_flavor` to determine which type of authentication the caller used. The service implementor may also wish to inspect the other fields of `rq_cred` if the type is not one of the types supported by the RPC package.

2. That the request's `rq_clntcred` field is either `NULL` or points to a well formed structure that corresponds to a supported type of authentication credentials. As only INTERACTIVE UNIX System type is currently supported, `rq_clntcred` could be cast to a pointer to an `authunix_parms` structure. If `rq_clntcred` is `NULL`, the service implementor may wish to inspect the other (opaque) fields of `rq_cred` in case the service knows about a new type of authentication that the RPC package does not know about.

The remote users' service example can be extended so that it computes results for all users except UID 16:

```
nuser(rqstp, transp)
     struct svc_req *rqstp;
     SVCXPRT *transp;
{
     struct authunix_parms *unix_cred;
     int uid;
     unsigned long nusers;

     /*
      * we don't care about authentication for the null procedure
      */
     if (rqstp->rq_proc == NULLPROC) {
          if (!svc_sendreply(transp, xdr_void, 0)) {
               fprintf(stderr, "couldn't reply to RPC call\n");
               exit(1);
          }
          return;
     }

     /*
      * now get the uid
      */
     switch (rqstp->rq_cred.oa_flavor) {
     case AUTH_UNIX:
          unix_cred = (struct authunix_parms *) rqstp->rq_clntcred;
          uid = unix_cred->aup_uid;
          break;
     case AUTH_NULL:
     default:
        svcerr_weakauth(transp);
        return;
     }
     switch (rqstp->rq_proc) {
     case RUSERSPROC_NUM:

        /*
         * make sure the caller is allowed to call this
         * procedure.
         */
        if (uid == 16) {
                svcerr_systemerr(transp);
                return;
        }
        /*
         * code here to compute the number of users
         * and put in variable nusers
         */
        if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
                fprintf(stderr, "couldn't reply to RPC call\n");
                exit(1);
        }
        return;
     default:
        svcerr_noproc(transp);
        return;
     }
}
```

A few things should be noted here. First, it is customary not to check the authentication parameters associated with the NULLPROC (procedure number 0). Second, if the authentication parameter's type is not suitable for a particular user's service,

`svcerr_weakauth()` should be called. Finally, the service protocol itself should return status for access denied; in the case of the above example, the protocol does not have such a status, so the service primitive `svcerr_systemerr()` is called instead.

The last point underscores the relationship between the RPC authentication package and the services; RPC deals only with authentication and not with individual services' access control. The services themselves must implement their own access control policies and reflect these policies as return statuses in their protocols.

4.5 More Examples

4.5.1 Versions

By convention, the first version number of program `FOO` is `FOOVERS_ORIG`, and the most recent version is `FOOVERS`. Suppose there is a new version of the `nuser` program that returns an `unsigned short` rather than a `long`. If this version was named `RUSERSVERS_SHORT`, then a server that wants to support both versions would do a double register.

```
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_ORIG, nuser,
    IPPROTO_TCP)) {
        fprintf(stderr, "couldn't register RUSER service\n");
        exit(1);
}
if (!svc_register(transp, RUSERSPROG, RUSERSVERS_SHORT, nuser,
    IPPROTO_TCP)) {
        fprintf(stderr, "couldn't register RUSER service\n");
        exit(1);
}
```

Both versions can be handled by the same C procedure:

```
nuser(rqstp, transp)
     struct svc_req *rqstp;
     SVCXPRT *transp;
{
     unsigned long nusers;
     unsigned short nusers2

     switch (rqstp->rq_proc) {
     case NULLPROC:
        if (!svc_sendreply(transp, xdr_void, 0)) {
             fprintf(stderr, "couldn't reply to RPC call\n");
             exit(1);
        }
        return;
     case RUSERSPROC_NUM:
        /*
         * code here to compute the number of users
         * and put in variable nusers
         */
        nusers2 = nusers;
        if (rqstp->rq_vers == RUSERSVERS_ORIG) {
             if (!svc_sendreply(transp, xdr_u_long, &nusers)) {
                 fprintf(stderr, "couldn't reply to RPC call\n");
                 exit(1);
             }
        }
        else {
             if (!svc_sendreply(transp, xdr_u_short, &nusers2)) {
                 fprintf(stderr, "couldn't reply to RPC call\n");
                 exit(1);
             }
        }
        return;
     default:
        svcerr_noproc(transp);
        return;
     }
}
```

4.5.2 TCP

Here is an example that is essentially `rcp`. The initiator of the RPC `snd()` call takes its standard input and sends it to the server `rcv()`, which prints it on standard output. The RPC call uses TCP. This also illustrates an XDR procedure that behaves differently on serialization than on deserialization:

```
/*
 * The xdr routine:
 *
 * on decode, read from wire, write onto fp
 * on encode, read from fp, write onto wire
 */
#include <stdio.h>
#include <rpc/rpc.h>

xdr_rcp(xdrs, fp)
     XDR *xdrs;
     FILE *fp;
{
     unsigned long size;
     char buf[MAXCHUNK], *p;

     if (xdrs->x_op == XDR_FREE) /* nothing to free */
             return (1);
     while (1) {
           if (xdrs->x_op == XDR_ENCODE) {
                 if ((size = fread(buf, sizeof(char), MAXCHUNK, fp))
                     == 0 && ferror(fp)) {
                          fprintf(stderr, "couldn't fread0);
                          exit(1);
                 }
           }
           p = buf;
           if (!xdr_bytes(xdrs, &p, &size, MAXCHUNK))
                   return (0);
           if (size == 0)
                   return (1);
           if (xdrs->x_op == XDR_DECODE) {
                 if (fwrite(buf, sizeof(char), size, fp) != size) {
                          fprintf(stderr, "couldn't fwrite0);
                          exit(1);
                 }
           }
     }
}
```

```
/*
 * The sender routines
 */
#include <stdio.h>
#include <netdb.h>
#include <rpc/rpc.h>
#include <sys/socket.h>
#include <sys/fs/nfs/time.h>

main(argc, argv)
     int argc;
     char **argv;
{
     int err;

     if (argc < 2) {
          fprintf(stderr, "usage: %s server-name\n", argv[0]);
          exit(-1);
     }
     if ((err = callrpctcp(argv[1], RCPPROG, RCPPROC_FP, RCPVERS,
        xdr_rcp, stdin, xdr_void, 0)) != 0) {
          clnt_perrno(err);
          fprintf(stderr, " couldn't make RPC call\n");
          exit(1);
     }
}

callrpctcp(host, prognum, procnum, versnum, inproc, in, outproc, out)
     char *host, *in, *out;
     xdrproc_t inproc, outproc;
{
     struct sockaddr_in server_addr;
     int socket = RPC_ANYSOCK;
     enum clnt_stat clnt_stat;
     struct hostent *hp;
     register CLIENT *client;
     struct timeval total_timeout;

     if ((hp = gethostbyname(host)) == NULL) {
          fprintf(stderr, "cannot get addr for '%s'\n", host);
          exit(-1);
     }
     bcopy(hp->h_addr, (caddr_t)&server_addr.sin_addr, hp->h_length);
     server_addr.sin_family = AF_INET;
     server_addr.sin_port =  0;
     if ((client = clnttcp_create(&server_addr, prognum,
        versnum, &socket, BUFSIZ, BUFSIZ)) == NULL) {
          perror("rpctcp_create");
          exit(-1);
     }
     total_timeout.tv_sec = 20;
     total_timeout.tv_usec = 0;
     clnt_stat = clnt_call(client, procnum, inproc, in,
        outproc, out, total_timeout);
     clnt_destroy(client)
     return ((int)clnt_stat);
}
```

```
/*
 * The receiving routines
 */
#include <stdio.h>
#include <rpc/rpc.h>

main()
{
     register SVCXPRT *transp;

     if ((transp = svctcp_create(RPC_ANYSOCK, 1024, 1024)) == NULL) {
             fprintf(stderr, "svctcp_create: error\n");
             exit(1);
     }
     pmap_unset(RCPPROG, RCPVERS);
     if (!svc_register(transp, RCPPROG, RCPVERS, rcp_service,
         IPPROTO_TCP)) {
             fprintf(stderr, "svc_register: error\n");
             exit(1);
     }
     svc_run();  /* never returns */
     fprintf(stderr, "svc_run should never return\n");
}

rcp_service(rqstp, transp)
     register struct svc_req *rqstp;
     register SVCXPRT *transp;
{
     switch (rqstp->rq_proc) {
     case NULLPROC:
             if (!svc_sendreply(transp, xdr_void, 0)) {
                     fprintf(stderr, "err: rcp_service");
                     exit(1);
             }
             return;
     case RCPPROC_FP:
             if (!svc_getargs(transp, xdr_rcp, stdout)) {
                     svcerr_decode(transp);
                     return;
             }
             if (!svc_sendreply(transp, xdr_void, 0)) {
                     fprintf(stderr, "can't reply\n");
                     return;
             }
             exit(0);
     default:
             svcerr_noproc(transp);
             return;
     }
}
```

4.5.3 Callback Procedures

Occasionally, it is useful to have a server become a client and make an RPC call back to the process which is its client. An example is remote debugging, where the client is a window system program, and the server is a debugger running on the remote machine. Most of the time, the user clicks a mouse button at the debugging window, which converts this to a debugger command, and then makes an RPC call to the server (where the debugger is actually running), telling it to execute that command. However, when the debugger hits a breakpoint, the roles are reversed, and the debugger wants to make an RPC call to the window program, so that it can inform the user that a breakpoint has been reached.

In order to do an RPC callback, a program number to make the RPC call on is needed. Since this will be a dynamically generated program number, it should be in the transient range, 0x40000000 – 0x5FFFFFFF. The routine `gettransient()` returns a valid program number in the transient range and registers it with the portmapper. It only talks to the portmapper running on the same machine as the `gettransient()` routine itself. The call to `pmap_set()` is a test and set operation, in that it indivisibly tests whether a program number has already been registered and if it has not, reserves it. On return, the `sockp` argument will contain a socket that can be used as the argument to an `svcudp_create()` or `svctcp_create()` call.

```
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/socket.h>

gettransient(proto, vers, sockp)
        int *sockp;
{
        static int prognum = 0x40000000;
        int s, len, socktype;
        struct sockaddr_in addr;

        switch(proto) {
        case IPPROTO_UDP:
                socktype = SOCK_DGRAM;
                break;
        case IPPROTO_TCP:
                socktype = SOCK_STREAM;
                break;
        default:
                fprintf(stderr, "unknown protocol type\n");
                return (0);
        }
        if (*sockp == RPC_ANYSOCK) {
                if ((s = socket(AF_INET, socktype, 0)) < 0) {
                        perror("socket");
                        return (0);
                }
                *sockp = s;
        }
        else
                s = *sockp;
        addr.sin_addr.s_addr = 0;
        addr.sin_family = AF_INET;
        addr.sin_port = 0;
        len = sizeof(addr);
        /*
         * may be already bound, so don't check for err
         */
        (void) bind(s, &addr, len);
        if (getsockname(s, &addr, &len) < 0) {
                perror("getsockname");
                return (0);
        }
        while (pmap_set(prognum++, vers, proto, addr.sin_port) == 0)
                continue;
        return (prognum-1);
}
```

The following pair of programs illustrate how to use the `gettransient()` routine. The client makes an RPC call to the server, passing it a transient program number. Then the client waits around to receive a callback from the server at that program number. The server registers the program **EXAMPLEPROG**, so that it can receive the RPC call informing it of the callback program number. Then at some random time (on receiving an **ALRM** signal in this example), it sends a callback RPC call, using the program number it received earlier:

```
/*
 * client
 */
#include <stdio.h>
#include <rpc/rpc.h>

int callback();
char hostname[256];

main(argc, argv)
     char **argv;
{
     int x, ans, s;
     SVCXPRT *xprt;

     gethostname(hostname, sizeof(hostname));
     s = RPC_ANYSOCK;
     x = gettransient(IPPROTO_UDP, 1, &s);
     fprintf(stderr, "client gets prognum %d\n", x);

     if ((xprt = svcudp_create(s)) == NULL) {
             fprintf(stderr, "rpc_server: svcudp_create\n");
             exit(1);
     }
     (void)svc_register(xprt, x, 1, callback, 0);

     ans = callrpc(hostname, EXAMPLEPROG, EXAMPLEPROC_CALLBACK,
         EXAMPLEVERS, xdr_int, &x, xdr_void, 0);
     if (ans != 0) {
             fprintf(stderr, "call: ");
             clnt_perrno(ans);
             fprintf(stderr, "\n");
     }
     svc_run();
     fprintf(stderr, "Error: svc_run shouldn't have returned\n");
}

callback(rqstp, transp)
     register struct svc_req *rqstp;
     register SVCXPRT *transp;
{
     switch (rqstp->rq_proc) {
     case 0:
             if (!svc_sendreply(transp, xdr_void, 0)) {
                     fprintf(stderr, "err: rusersd\n");
                     exit(1);
             }
             exit(0);
     case 1:
             if (!svc_getargs(transp, xdr_void, 0)) {
                     svcerr_decode(transp);
                     exit(1);
             }
             fprintf(stderr, "client got callback\n");
             if (!svc_sendreply(transp, xdr_void, 0)) {
                     fprintf(stderr, "err: rusersd");
                     exit(1);
             }
     }
}
```

```
/*
 * server
 */
#include <stdio.h>
#include <rpc/rpc.h>
#include <sys/signal.h>

char *getnewprog();
char hostname[256];
int docallback();
int pnum;                /*program number for callback routine */

main(argc, argv)
     char **argv;
{
     gethostname(hostname, sizeof(hostname));
     registerrpc(EXAMPLEPROG, EXAMPLEPROC_CALLBACK, EXAMPLEVERS,
         getnewprog, xdr_int, xdr_void);
     fprintf(stderr, "server going into svc_run\n");
     alarm(10);
     signal(SIGALRM, docallback);
     svc_run();
     fprintf(stderr, "Error: svc_run shouldn't have returned\n");
}

char *
getnewprog(pnump)
     char *pnump;
{
     pnum = *(int *)pnump;
     return (NULL);
}

docallback()
{
     int ans;

     ans = callrpc(hostname, pnum, 1, 1, xdr_void, 0, xdr_void, 0);
     if (ans != 0) {
             fprintf(stderr, "server: ");
             clnt_perrno(ans);
             fprintf(stderr, "\n");
     }
}
```

4.6 Synopsis of RPC Routines

auth_destroy()

```
void
auth_destroy(auth)
        AUTH *auth;
```

A macro that destroys the authentication information associated with `auth`. Destruction usually involves deallocation of private data structures. The use of `auth` is undefined after calling `auth_destroy()`.

authnone_create( )

```
AUTH *
authnone_create()
```

Creates and returns an RPC authentication handle that passes no usable authentication information with each remote procedure call.

authunix_create( )

```
AUTH *
authunix_create(host, uid, gid, len, aup_gids)
        char *host;
        int uid, gid, len, *aup_gids;
```

Creates and returns an RPC authentication handle that contains INTERACTIVE UNIX System authentication information. The parameter `host` is the name of the machine on which the information was created; `uid` is the user's user ID; `gid` is the user's current group ID; and `len` and `aup_gids` refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

authunix_create_default( )

```
AUTH *
authunix_create_default()
```

Calls `authunix_create( )` with the appropriate parameters.

callrpc( )

```
callrpc(host, prognum, versnum, procnum, inproc, in, outproc,
out)
        char *host;
        ulong prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
```

Calls the remote procedure associated with `prognum`, `versnum`, and `procnum` on the machine, `host`. The parameter `in` is the address of the procedure's argument(s), and `out` is the address of where to place the result(s); `inproc` is used to encode the procedure's parameters, and `outproc` is used to decode the procedure's results. This routine returns 0 if it succeeds, or the value of `enum clnt_stat` cast to an integer if it fails. The routine `clnt_perrno( )` is handy for translating failure statuses into messages.

WARNING: Calling remote procedures with this routine uses UDP/IP as a transport; see `clntudp_create()` for restrictions.

clnt_broadcast()

```
enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in,
outproc, out, eachresult)
        ulong prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
        resultproc_t eachresult;
```

Like `callrpc()`, except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls `eachresult`, the form of which is:

```
eachresult(out, addr)
        char *out;
        struct sockaddr_in *addr;
```

where `out` is the same as `out` passed to `clnt_broadcast()`, except that the remote procedure's output is decoded there; `addr` points to the address of the machine that sent the results. If `eachresult()` returns 0, `clnt_broadcast()` waits for more replies; otherwise it returns with appropriate status.

clnt_call()

```
enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
        CLIENT *clnt; long procnum;
        xdrproc_t inproc, outproc;
        char *in, *out;
        struct timeval tout;
```

A macro that calls the remote procedure `procnum` associated with the client handle, `clnt`, which is obtained with an RPC client creation routine such as `clntudp_create`. The parameter `in` is the address of the procedure's argument(s), and `out` is the address where the result(s) should be placed; `inproc` is used to encode the procedure's parameters, and `outproc` is used to decode the procedure's results; and `tout` is the time allowed for results to come back.

clnt_destroy()

```
clnt_destroy(clnt)
        CLIENT *clnt;
```

A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including `clnt` itself. Use of `clnt` is undefined after calling `clnt_destroy()`.

WARNING: Client destruction routines do not close sockets associated with `clnt`; this is the responsibility of the user.

clnt_freeres()

```
clnt_freeres(clnt, outproc, out)
        CLIENT *clnt;
        xdrproc_t outproc;
        char *out;
```

A macro that frees any data allocated by the RPC/XDR system when it decodes the results of an RPC call. The parameter `out` is the address of the results, and `outproc` is the XDR routine describing the results in simple primitives. This routine returns 1 if the results were successfully freed, 0 otherwise.

clnt_geterr()

```
void
clnt_geterr(clnt, errp)
        CLIENT *clnt;
        struct rpc_err *errp;
```

A macro that copies the error structure out of the client handle to the structure at address `errp`.

clnt_pcreateerror()

```
void
clnt_pcreateerror(s)
        char *s;
```

Prints a message to standard error indicating why a client RPC handle could not be created. The message is prepended with string `s` and a colon.

clnt_perrno()

```
void
clnt_perrno(stat)
        enum clnt_stat stat;
```

Prints a message to standard error corresponding to the condition indicated by `stat`.

clnt_perror()

```
clnt_perror(clnt, s)
        CLIENT *clnt;
        char *s;
```

Prints a message to standard error indicating why an RPC call failed; `clnt` is the handle used to do the call. The message is prepended with string `s` and a colon.

clntraw_create()

```
CLIENT *
clntraw_create(prognum, versnum)
        ulong prognum, versnum;
```

This routine creates a toy RPC client for the remote program `prognum`, version `versnum`. The transport used to pass messages to the service is actually a buffer within the process's address space, so the corresponding RPC server should live in the same address space; see `svcraw_create()`. This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns `NULL` if it fails.

clnttcp_create()

```
CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
        struct sockaddr_in *addr;
        ulong prognum, versnum;
        int *sockp;
        uint sendsz, recvsz;
```

This routine creates an RPC client for the remote program `prognum`, version `versnum`; the client uses TCP/IP as a transport. The remote program is located at Internet address `*addr`. If `addr->sin_port` is 0, then it is set to the actual port that the remote program is listening on (the remote `portmap` service is consulted for this information). The parameter `*sockp` is a socket; if it is `RPC_ANYSOCK`, then this routine opens a new one and sets `*sockp`. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive buffers with the

parameters `sendsz` and `recvsz`; values of 0 choose suitable defaults. This routine returns `NULL` if it fails.

clntudp_create( )

```
CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
        struct sockaddr_in *addr;
        ulong prognum, versnum;
        struct timeval wait;
        int *sockp;
```

This routine creates an RPC client for the remote program `prognum`, version `versnum`; the client uses UDP/IP as a transport. The remote program is located at Internet address `*addr`. If `addr->sin_port` is 0, then it is set to the actual port that the remote program is listening on (the remote `portmap` service is consulted for this information). The parameter `*sockp` is a socket; if it is `RPC_ANYSOCK`, then this routine opens a new one and sets `*sockp`. The UDP transport resends the call message in intervals of `wait` time until a response is received or until the call times out.

WARNING: Since UDP-based RPC messages can only hold up to 4 or 8 KB of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

get_myaddress( )

```
void
get_myaddress(addr)
        struct sockaddr_in *addr;
```

Places the machine's IP address in `*addr`, without consulting the library routines that deal with host name to address resolution. The port number is always set to `htons(PMAPPORT)`.

pmap_getmaps( )

```
struct pmaplist *
pmap_getmaps(addr)
        struct sockaddr_in *addr;
```

A user interface to the `portmap` service, which returns a list of the current RPC program-to-port mappings on the host located at IP address `*addr`. This routine can return `NULL`. The command `rpcinfo -p` uses this routine.

pmap_getport()

```
ushort
pmap_getport(addr, prognum, versnum, protocol)
        struct sockaddr_in *addr;
        ulong prognum, versnum, protocol;
```

A user interface to the `portmap` service, which returns the port number on which waits a service that supports program number `prognum`, version `versnum`, and speaks the transport protocol associated with protocol. A return value of 0 means that the mapping does not exist or that the RPC system failed to contact the remote `portmap` service. In the latter case, the global variable `rpc_createerr` contains the RPC status.

pmap_rmtcall()

```
enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum,
          inproc, in, outproc, out, tout, portp)
        struct sockaddr_in *addr;
        ulong prognum, versnum, procnum;
        char *in, *out;
        xdrproc_t inproc, outproc;
        struct timeval tout;
        ulong *portp;
```

A user interface to the `portmap` service, which instructs `portmap` on the host at IP address `*addr` to make an RPC call on the user's behalf to a procedure on that host. The parameter `*portp` will be modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in `callrpc()` and `clnt_call()`; refer also to `clnt_broadcast()`.

pmap_set()

```
pmap_set(prognum, versnum, protocol, port)
        ulong prognum, versnum, protocol;
        ushort port;
```

A user interface to the `portmap` service, which establishes a mapping between the [`prognum, versnum, protocol`] triple and `port` on the machine's `portmap` service. The value of protocol is most likely `IPPROTO_UDP` or `IPPROTO_TCP`. This routine returns 1 if it succeeds, 0 otherwise.

pmap_unset( )

```
pmap_unset(prognum, versnum)
        ulong prognum, versnum;
```

A user interface to the `portmap` service, which destroys all mappings between the triple `[prognum,versnum,*]` and `ports` on the machine's `portmap` service. This routine returns 1 if it succeeds, 0 otherwise.

registerrpc( )

```
registerrpc(prognum, versnum, procnum, procname, inproc,
outproc)
        ulong prognum, versnum, procnum;
        char *(*procname)();
        xdrproc_t inproc, outproc;
```

Registers procedure `procname` with the RPC service package. If a request arrives for program `prognum`, version `versnum`, and procedure `procnum`, `procname` is called with a pointer to its parameter(s); `procname` should return a pointer to its static result(s); `inproc` is used to decode the parameters, while `outproc` is used to encode the results. This routine returns 0 if the registration succeeded, -1 otherwise.

WARNING: Remote procedures registered in this form are accessed using the UDP/IP transport; refer to `svcudp_create( )` for restrictions.

rpc_createerr

```
struct rpc_createerr rpc_createerr;
```

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine `clnt_pcreateerror( )` to print the reason why.

svc_destroy( )

```
svc_destroy(xprt)
        SVCXPRT *xprt;
```

A macro that destroys the RPC service transport handle `xprt`. Destruction usually involves deallocation of private data structures, including `xprt` itself. Use of `xprt` is undefined after calling this routine.

svc_fds

```
int svc_fds;
```

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the `select` system call. This is only of interest if a service implementor does not call `svc_run()`, but rather does his own asynchronous event processing. This variable is read-only, yet it may change after calls to `svc_getreq()` or any creation routines.

svc_freeargs()

```
svc_freeargs(xprt, inproc, in)
        SVCXPRT *xprt;
        xdrproc_t inproc;
        char *in;
```

A macro that frees any data allocated by the RPC/XDR system when it decodes the arguments to a service procedure using `svc_getargs()`. This routine returns 1 if the results were successfully freed, 0 otherwise.

svc_getargs()

```
svc_getargs(xprt, inproc, in)
        SVCXPRT *xprt;
        xdrproc_t inproc;
        char *in;
```

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle `xprt`. The parameter `in` is the address where the arguments will be placed; `inproc` is the XDR routine used to decode the arguments. This routine returns 1 if decoding succeeds, 0 otherwise.

svc_getcaller( )

```
struct sockaddr_in
svc_getcaller(xprt)
        SVCXPRT xprt;
```

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle `xprt`.

svc_getreq( )

```
svc_getreq(rdfds)
        int rdfds;
```

This routine is only of interest if a service implementor does not call `svc_run( )`, but instead implements custom asynchronous event processing. It is called when the `select` system call has determined that an RPC request has arrived on some RPC socket(s); `rdfds` is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of `rdfds` have been serviced.

svc_register( )

```
svc_register(xprt, prognum, versnum, dispatch, protocol)
        SVCXPRT *xprt;
        ulong prognum, versnum;
        void (*dispatch)();
        ulong protocol;
```

Associates `prognum` and `versnum` with the service dispatch procedure `dispatch`. If `protocol` is not zero, a mapping of `[prognum, versnum, protocol]` to `xprt->xp_port` is also established with the local `portmap` service (generally `protocol` is 0, `IPPROTO_UDP`, or `IPPROTO_TCP`). The procedure `dispatch( )` has the following form:

```
dispatch(request, xprt)
        struct svc_req *request;
        SVCXPRT *xprt;
```

The `svc_register` routine returns 1 if it succeeds, 0 otherwise.

svc_run()

```
svc_run()
```

This routine never returns. It waits for RPC requests to arrive and calls the appropriate service procedure (using `svc_getreq`) when one arrives. This procedure is usually waiting for a `select` system call to return.

svc_sendreply()

```
svc_sendreply(xprt, outproc, out)
        SVCXPRT *xprt;
        xdrproc_t outproc;
        char *out;
```

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter `xprt` is the caller's associated transport handle, `outproc` is the XDR routine that is used to encode the results, and `out` is the address of the results. This routine returns 1 if it succeeds, 0 otherwise.

svc_unregister()

```
void
svc_unregister(prognum, versnum)
        ulong prognum, versnum;
```

Removes all mapping of the double `[prognum, versnum]` to dispatch routines and of the triple `[prognum, versnum, *]` to port number.

svcerr_auth()

```
void
svcerr_auth(xprt, why)
        SVCXPRT *xprt;
        enum auth_stat why;
```

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

svcerr_decode( )

```
void
svcerr_decode(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that cannot successfully decode its parameters. See also `svc_getargs( )`.

svcerr_noproc( )

```
void
svcerr_noproc(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that does not implement the desired procedure number the caller requested.

svcerr_noprog( )

```
void
svcerr_noprog(xprt)
        SVCXPRT *xprt;
```

Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

svcerr_progvers( )

```
void
svcerr_progvers(xprt)
        SVCXPRT *xprt;
```

Called when the desired version of a program is not registered with the RPC package. Service implementors usually do not need this routine.

svcerr_systemerr( )

```
void
svcerr_systemerr(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine when it detects a system error not covered by any particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

svcerr_weakauth()

```
void
svcerr_weakauth(xprt)
        SVCXPRT *xprt;
```

Called by a service dispatch routine that refuses to perform a remote procedure call because of insufficient (although correct) authentication parameters. The routine calls `svcerr_auth(xprt,AUTH_TOOWEAK)`.

svcraw_create()

```
SVCXPRT *
svcraw_create()
```

This routine creates a toy RPC service transport, to which it returns a pointer. Since the transport is really a buffer within the process's address space, the corresponding RPC client should live in the same address space; see `clntraw_create()`. This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times) without any kernel interference. This routine returns `NULL` if it fails.

svctcp_create()

```
SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
        int sock;
        uint send_buf_size, recv_buf_size;
```

This routine creates a TCP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket `sock`, which may be `RPC_ANYSOCK`, in which case a new socket is created. If the socket is not bound to a local TCP port, then this routine binds it to an arbitrary port. Upon completion, `xprt->xp_sock` is the transport's socket number, and `xprt->xp_port` is the transport's port number. This routine returns `NULL` if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of the `send` and `receive` buffers; values of 0 choose suitable defaults.

svcudp_create( )

```
SVCXPRT *
svcudp_create(sock)
        int sock;
```

This routine creates a UDP/IP-based RPC service transport, to which it returns a pointer. The transport is associated with the socket `sock`, which may be `RPC_ANYSOCK`, in which case a new socket is created. If the socket is not bound to a local UDP port, this routine binds it to an arbitrary port. Upon completion, `xprt->xp_sock` is the transport's socket number, and `xprt->xp_port` is the transport's port number. This routine returns `NULL` if it fails.

WARNING: Since UDP-based RPC messages can only hold up to 4 or 8 KB of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

xdr_accepted_reply( )

```
xdr_accepted_reply(xdrs, ar)
        XDR *xdrs;
        struct accepted_reply *ar;
```

Used for describing RPC messages externally. This routine is useful for users who wish to generate RPC-type messages without using the RPC package.

xdr_array( )

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
        XDR *xdrs;
        char **arrp;
        uint *sizep, maxsize, elsize;
        xdrproc_t elproc;
```

A filter primitive that translates between arrays and their corresponding external representations. The parameter `arrp` is the address of the pointer to the array, while `sizep` is the address of the element count of the array; this element count cannot exceed `maxsize`. The parameter `elsize` is the `sizeof( )` each of the array's elements, and `elproc` is an XDR filter that translates between the array elements' C form and their external representation. This routine returns 1 if it succeeds, 0 otherwise.

xdr_authunix_parms()

```
xdr_authunix_parms(xdrs, aupp)
        XDR *xdrs;
        struct authunix_parms *aupp;
```

Used for describing UNIX Operating System credentials externally. This routine is helpful for users who wish to generate these credentials without using the RPC authentication package.

xdr_bool()

```
xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either 1 or 0. This routine returns 1 if it succeeds, 0 otherwise.

xdr_bytes()

```
xdr_bytes(xdrs, sp, sizep, maxsize)
        XDR *xdrs;
        char **sp;
        uint *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter `sp` is the address of the string pointer. The length of the string is located at address `sizep`; strings cannot be longer than `maxsize`. This routine returns 1 if it succeeds, 0 otherwise.

xdr_callhdr()

```
void
xdr_callhdr(xdrs, chdr)
        XDR *xdrs;
        struct rpc_msg *chdr;
```

Used for describing RPC messages externally. This routine is helpful for users who wish to generate RPC-type messages without using the RPC package.

xdr_callmsg()

```
xdr_callmsg(xdrs, cmsg)
        XDR *xdrs;
        struct rpc_msg *cmsg;
```

Used for describing RPC messages externally. This routine is helpful for users who wish to generate RPC-type messages without using the RPC package.

xdr_double()

```
xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

A filter primitive that translates between C `double` precision floating point numbers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_enum()

```
xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;
```

A filter primitive that translates between C `enums` (actually integers) and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_float()

```
xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;
```

A filter primitive that translates between C `floats` (single precision floating point numbers) and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_inline()

```
long *
xdr_inline(xdrs, len)
        XDR *xdrs;
        int len;
```

A macro that invokes the in-line routine associated with the XDR stream `xdrs`. The routine returns a pointer to a contiguous piece of the stream's buffer; `len` is the byte length of the desired buffer. Note that pointer is cast to `long*`.

WARNING: `xdr_inline()` may return 0 (`NULL`) if it cannot allocate a contiguous piece of a buffer. Therefore, the

behavior may vary among stream instances; it exists for the sake of efficiency.

xdr_int()

```
xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_long()

```
xdr_long(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

A filter primitive that translates between C `long` integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_opaque()

```
xdr_opaque(xdrs, cp, cnt)
        XDR *xdrs;
        char *cp;
        uint cnt;
```

A filter primitive that translates between fixed size opaque data and its external representation. The parameter `cp` is the address of the opaque object, and `cnt` is its size in bytes. This routine returns 1 if it succeeds, 0 otherwise.

xdr_opaque_auth()

```
xdr_opaque_auth(xdrs, ap)
        XDR *xdrs;
        struct opaque_auth *ap;
```

Used for describing RPC messages externally. This routine is helpful for users who wish to generate RPC-type messages without using the RPC package.

xdr_pmap( )

```
xdr_pmap(xdrs, regs)
        XDR *xdrs;
        struct pmap *regs;
```

Used for describing parameters to various `portmap` procedures externally. This routine is helpful for users who wish to generate these parameters without using the `pmap` interface.

xdr_pmaplist( )

```
xdr_pmaplist(xdrs, rp)
        XDR *xdrs;
        struct pmaplist **rp;
```

Used for describing a list of port mappings externally. This routine is helpful for users who wish to generate these parameters without using the `pmap` interface.

xdr_reference( )

```
xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        uint size;
        xdrproc_t proc;
```

A primitive that provides pointer chasing within structures. The parameter `pp` is the address of the pointer, `size` is the `sizeof( )` the structure that `*pp` points to, and `proc` is an XDR procedure that filters the structure between its C form and its external representation. This routine returns 1 if it succeeds, 0 otherwise.

xdr_rejected_reply( )

```
xdr_rejected_reply(xdrs, rr)
        XDR *xdrs;
        struct rejected_reply *rr;
```

Used for describing RPC messages externally. This routine is helpful for users who wish to generate RPC-type messages without using the RPC package.

xdr_replymsg()

```
xdr_replymsg(xdrs, rmsg)
        XDR *xdrs;
        struct rpc_msg *rmsg;
```

Used for describing RPC messages externally. This routine is helpful for users who wish to generate RPC-type messages without using the RPC package.

xdr_short()

```
xdr_short(xdrs, sp)
        XDR *xdrs;
        short *sp;
```

A filter primitive that translates between C `short` integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_string()

```
xdr_string(xdrs, sp, maxsize)
        XDR *xdrs;
        char **sp;
        uint maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than `maxsize`. Note that `sp` is the address of the string's pointer. This routine returns 1 if it succeeds, 0 otherwise.

xdr_u_int()

```
xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;
```

A filter primitive that translates between C `unsigned` integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_u_long( )

```
xdr_u_long(xdrs, ulp)
        XDR *xdrs;
        unsigned long *ulp;
```

A filter primitive that translates between C `unsigned long` integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_u_short( )

```
xdr_u_short(xdrs, usp)
        XDR *xdrs;
        unsigned short *usp;
```

A filter primitive that translates between C `unsigned short` integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_union( )

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
        XDR *xdrs;
        int *dscmp;
        char *unp;
        struct xdr_discrim *choices;
        xdrproc_t dfault;
```

A filter primitive that translates between a discriminated C `union` and its corresponding external representation. The parameter `dscmp` is the address of the union's discriminant, while `unp` in the address of the union. This routine returns 1 if it succeeds, 0 otherwise.

xdr_void( )

```
xdr_void()
```

This routine always returns 1.

xdr_wrapstring()

```
xdr_wrapstring(xdrs, sp)
        XDR *xdrs;
        char **sp;
```

The primitive `xdr_wrapstring()` calls `xdr_string(xdrs, sp,MAXUNSIGNED)`, where `MAXUNSIGNED` is the maximum value of an unsigned integer. This is useful because the RPC package passes only two parameters to XDR routines, whereas `xdr_string()`, one of the most frequently used primitives, requires three parameters. This routine returns 1 if it succeeds, 0 otherwise.

xprt_register()

```
void
xprt_register(xprt)
        SVCXPRT *xprt;
```

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable `svc_fds`. Service implementors usually do not need this routine.

xprt_unregister()

```
void
xprt_unregister(xprt)
        SVCXPRT *xprt;
```

Before an RPC service transport handle is destroyed, it should deregister itself with the RPC service package. This routine modifies the global variable `svc_fds`. Service implementors usually do not need this routine.

5. RPC PROTOCOL SPECIFICATION

5.1 Introduction

This section specifies a message protocol used in implementing the Remote Procedure Call (RPC) package. The message protocol is specified with the eXternal Data Representation (XDR) language.

This section assumes that you are familiar with both RPC and XDR. It does not attempt to justify RPC or its uses. Also, the casual user of RPC does not need to be familiar with the information in this section.

5.1.1 Terminology

This section discusses servers, services, programs, procedures, clients, and versions. A server is a machine where some number of network services are implemented. A service is a collection of one or more remote programs. A remote program implements one or more remote procedures; the procedures, their parameters, and results are documented in the specific program's protocol specification (see section 5.5 for an example). Network clients are pieces of software that initiate remote procedure calls to services. A server may support more than one version of a remote program in order to be forward compatible with changing protocols.

For example, a network file service may be composed of two programs. One program may deal with high-level applications such as file system access control and locking. The other may deal with low-level file I/O and have procedures like "read" and "write." A client machine of the network file service would call the procedures associated with the two programs of the service on behalf of some user on the client machine.

5.1.2 The RPC Model

The remote procedure call model is similar to the local procedure call model. In the local case, the caller places arguments to a procedure in some well specified location (such as a result register). It then transfers control to the procedure, and eventually gains back control. At that point, the results of the procedure are extracted from the well specified location, and the caller continues execution.

The remote procedure call is similar, except that one thread of control winds through two processes – one is the caller's process, the other is a server's process. That is, the caller process sends a call message to the server process and waits (blocks) for a reply

message. The contents of the call message include the procedure's parameters. The contents of the reply message include the procedure's results. Once the reply message is received, the results of the procedure are extracted, and the caller's execution is resumed.

On the server side, a process is dormant awaiting the arrival of a call message. When one arrives, the server process extracts the procedure's parameters, computes the results, sends a reply message, and then awaits the next call message. Note that in this model, only one of the two processes is active at any given time. That is, the RPC protocol does not explicitly support multi-threading of caller or server processes.

5.1.3 Transports and Semantics

The RPC protocol is independent of transport protocols. That is, RPC does not care how a message is passed from one process to another. The protocol only deals with the specification and interpretation of messages.

Because of transport independence, the RPC protocol does not attach specific semantics to the remote procedures or their execution. Some semantics can be inferred from (but should be explicitly specified by) the underlying transport protocol. For example, RPC message passing using UDP/IP is unreliable. Thus, if the caller retransmits call messages after short time-outs, the only thing which can be inferred from no reply message is that the remote procedure was executed zero or more times (and from a reply message, one or more times). On the other hand, RPC message passing using TCP/IP is reliable. No reply message means that the remote procedure was executed at most once, whereas a reply message means that the remote procedure was executed exactly once.

☛ Note that RPC is currently implemented on top of TCP/IP and UDP/IP transports.

5.1.4 Binding and Rendezvous Independence

The act of binding a client to a service is *not* part of the remote procedure call specification. This important and necessary function is left up to some higher level software. (The software may use RPC itself; see section 5.7)

Implementors should think of the RPC protocol as the jump-subroutine instruction ("JSR") of a network; the loader (binder)

makes JSR useful, and the loader itself uses JSR to accomplish its task. Likewise, the network makes RPC useful, using RPC to accomplish this task.

5.1.5 Message Authentication

The RPC protocol provides the fields necessary for a client to identify itself to a service and vice versa. Security and access control mechanisms can be built on top of the message authentication.

5.2 Requirements

The RPC protocol must provide for the following:

- Unique specification of a procedure to be called
- Provisions for matching response messages to request messages
- Provisions for authenticating the caller to service and vice versa

Besides these requirements, features that detect the following are worth supporting because of protocol roll-over errors, implementation bugs, user error, and network administration:

- RPC protocol mismatches
- Remote program protocol version mismatches
- Protocol errors (like mis-specification of a procedure's parameters)
- Reasons why remote authentication failed
- Any other reasons why the desired procedure was not called

5.2.1 Remote Programs and Procedures

The RPC call message has three unsigned fields: remote program number, remote program version number, and remote procedure number. The three fields uniquely identify the procedure to be called. Program numbers are administered by some central authority (currently Sun Microsystems). Once an implementor has a program number, he can implement his remote program; the first implementation would most probably have the version number of 1. Because most new protocols evolve into better, stable, and mature protocols, a version field of the call message identifies which version of the protocol the caller is using. Version numbers make speaking old and new protocols through the same server process possible.

The procedure number identifies the procedure to be called. These numbers are documented in the specific program's protocol specification. For example, a file service's protocol specification may state that its procedure number 5 is `read` and procedure number 12 is `write`.

Just as remote program protocols may change over several versions, the actual RPC message protocol could also change. Therefore, the call message also has the RPC version number in it; this field must be two (2).

The reply message to a request message has enough information to distinguish the following error conditions:

- The remote implementation of RPC does not speak protocol version 2. The lowest and highest supported RPC version numbers are returned.
- The remote program is not available on the remote system.
- The remote program does not support the requested version number. The lowest and highest supported remote program version numbers are returned.
- The requested procedure number does not exist. (This is usually a caller side protocol or programming error.)
- The parameters to the remote procedure appear to be garbage from the server's point of view. (Again, this is caused by a disagreement about the protocol between client and service.)

5.2.2 Authentication

Provisions for authentication of caller to service and vice versa are provided as a part of the RPC protocol. The call message has two authentication fields: the credentials and verifier. The reply message has one authentication field: the response verifier. The RPC protocol specification defines all three fields to be the following opaque type:

```
enum auth_flavor {
        AUTH_NULL       = 0,
        AUTH_UNIX       = 1,
        AUTH_SHORT      = 2
        /* and more to be defined */
};
struct opaque_auth {
        union switch (enum auth_flavor) {
                default: string auth_body<400>;
        };
};
```

Any `opaque_auth` structure is an `auth_flavor` enumeration followed by a counted string, whose bytes are opaque to the RPC protocol implementation.

The interpretation and semantics of the data contained within the authentication fields is specified by individual, independent authentication protocol specifications. Section 5.5 defines three authentication protocols.

If authentication parameters are rejected, the response message contains information stating why they are rejected.

5.2.3 Program Number Assignment

Program numbers are given out in groups of 0x20000000 (536870912) according to the following chart:

0	-	1fffffff	defined by Sun
20000000	-	3fffffff	defined by user
40000000	-	5fffffff	transient
60000000	-	7fffffff	reserved
80000000	-	9fffffff	reserved
a0000000	-	bfffffff	reserved
c0000000	-	dfffffff	reserved
e0000000	-	ffffffff	reserved

The first group is a range of numbers administered by Sun Microsystems and should be identical for all RPC users. When a user develops an application that might be of general interest, that application should be given an assigned number in the first range. The second range is for applications peculiar to a particular user; it is intended primarily for debugging new programs. The third group is for applications that generate program numbers dynamically. The final groups are reserved for future use and should not be used.

5.3 Other Uses and Abuses of the RPC Protocol

The intended use of this protocol is for calling remote procedures. That is, each call message is matched with a response message. However, the protocol itself is a message passing protocol with which other (non-RPC) protocols can be implemented. The RPC message protocol is currently used for two non-RPC protocols: batching (or pipelining) and broadcast RPC. These two protocols are discussed, but not defined, below.

5.3.1 Batching

Batching allows a client to send an arbitrarily large sequence of call messages to a server; batching uses reliable bytes stream protocols (like TCP/IP) for their transport. In the case of batching, the client never waits for a reply from the server, and the server does not send replies to batch requests. A sequence of batch calls is usually terminated by a legitimate RPC in order to flush the pipeline (with positive acknowledgment).

5.3.2 Broadcast RPC

In broadcast RPC-based protocols, the client sends a broadcast packet to the network and waits for numerous replies. Broadcast RPCs use unreliable, packet-based protocols, such as UDP/IP, as their transports. Servers that support broadcast protocols only respond when the request is successfully processed, and are silent in the face of errors.

5.4 The RPC Message Protocol

This section defines the RPC message protocol in the XDR data description language. The message is defined in a top down style. Note that this is an XDR specification, not C code.

```
enum msg_type {
        CALL = 0,
        REPLY = 1
};

/*
 * A reply to a call message can take on two forms:
 * the message was either accepted or rejected.
 */
enum reply_stat {
        MSG_ACCEPTED = 0,
        MSG_DENIED = 1
};

/*
 * Given that a call message was accepted,
 * the following is the status of
 * an attempt to call a remote procedure.
 */
enum accept_stat {
        SUCCESS = 0,
                /* remote procedure was successfully executed */
        PROG_UNAVAIL = 1,
                /* remote machine exports the program number */
        PROG_MISMATCH = 2,
                /* remote machine can't support version number */
        PROC_UNAVAIL = 3,
                /* remote program doesn't know about procedure */
        GARBAGE_ARGS = 4
                /* remote procedure can't figure out parameters */
};

/*
 * Reasons why a call message was rejected:
 */
enum reject_stat {
        RPC_MISMATCH = 0,
                /* RPC version number was not two (2) */
        AUTH_ERROR = 1
                /* caller not authenticated on remote machine */
};
```

```
/*
 * Why authentication failed:
 */
enum auth_stat {
        AUTH_BADCRED = 1,
                /* bogus credentials (seal broken) */
        AUTH_REJECTEDCRED = 2,
                /* client should begin new session */
        AUTH_BADVERF = 3,
                /* bogus verifier (seal broken) */
        AUTH_REJECTEDVERF = 4,
                /* verifier expired or was replayed */
        AUTH_TOOWEAK = 5
                /* rejected due to security reasons */
};

/*
 * The RPC message:
 * All messages start with a transaction identifier,
 * xid, followed by a two-armed discriminated union.
 * The union's discriminant is a msg_type which
 * switches to one of the two types of the message.
 * The xid of a REPLY message always matches that
 * of the initiating CALL message.
 * NB: The xid field is only used for clients
 * matching reply messages with call messages;
 * the service side cannot treat this ID as any
 * type of sequence number.
 */
struct rpc_msg {
        unsigned xid;
        union switch (enum msg_type) {
                CALL: struct call_body;
                REPLY: struct reply_body;
        };
};

/*
 * Body of an RPC request call:
 * In version 2 of the RPC protocol specification,
 * rpcvers must be equal to 2.
 * The fields prog, vers, and proc specify the
 * remote program, its version, and the procedure
 * within the remote program to be called.
 * These fields are followed by two authentication
 * parameters, cred (authentication credentials)
 * and verf (authentication verifier).  The
 * authentication parameters are followed by
 * the parameters to the remote procedure;
 * these parameters are specified by the
 * specific program protocol.
 */
struct call_body {
        unsigned rpcvers;        /* must be equal to two (2) */
        unsigned prog;
        unsigned vers;
        unsigned proc;
        struct opaque_auth cred;
        struct opaque_auth verf;
        /* procedure specific parameters start here */
};
```

```
/*
 * Body of a reply to an RPC request.
 * The call message was either accepted or rejected.
 */
struct reply_body {
        union switch (enum reply_stat) {
                MSG_ACCEPTED: struct accepted_reply;
                MSG_DENIED: struct rejected_reply;
        };
};

/*
 * Reply to an RPC request that was accepted by the server.
 * Note: there could be an error even though the request
 * was accepted.  The first field is an authentication
 * verifier which the server generates in order to
 * validate itself to the caller.  It is followed by
 * a union whose discriminant is an enum accept_stat.
 * The SUCCESS arm of the union is protocol specific.
 * The PROG_UNAVAIL, PROC_UNAVAIL, and GARBAGE_ARGS
 * arms of the union are void.  The PROG_MISMATCH
 * arm specifies the lowest and highest version
 * numbers of the remote program that are supported
 * by the server.
 */
struct accepted_reply {
        struct opaque_auth verf;
        union switch (enum accept_stat) {
                SUCCESS: struct {
                        /*
                         * procedure-specific results start here
                         */
                };
                PROG_MISMATCH: struct {
                        unsigned low;
                        unsigned high;
                };
                default: struct {
                        /*
                         * void. Cases include PROG_UNAVAIL,
                         * PROC_UNAVAIL, and GARBAGE_ARGS.
                         */
                };
        };
};
```

```
/*
 * Reply to an RPC request that was rejected by the server.
 * The request can be rejected because of two reasons -
 * either the server is not running a compatible version
 * of the RPC protocol (RPC_MISMATCH), or the server
 * refused to authenticate the caller (AUTH_ERROR).
 * In the case of an RPC version mismatch, the server
 * returns the lowest and highest supported RPC version
 * numbers. In the case of refused authentication,
 * the failure status is returned.
 */
struct rejected_reply {
        union switch (enum reject_stat) {
                RPC_MISMATCH: struct {
                        unsigned low;
                        unsigned high;
                };
                AUTH_ERROR: enum auth_stat;
        };
};
```

5.5 Authentication Parameter Specification

As previously stated, authentication parameters are opaque, but open-ended to the rest of the RPC protocol. This section defines some types of authentication that have been implemented and are generally supported.

5.5.1 Null Authentication

Often calls must be made in which the caller does not know who he is and the server does not care who the caller is. In this case, the `auth_flavor` value (the discriminant of the `opaque_auth`'s union) of the RPC message's credentials, verifier, and response verifier is `AUTH_NULL`(0). The bytes of the `auth_body` string are undefined. It is recommended that the string length be zero.

5.5.2 UNIX System Authentication

The caller of a remote procedure may wish to identify himself as he is identified on an INTERACTIVE UNIX Operating System. The value of the `credential`'s discriminant of an RPC call message is `AUTH_UNIX`(1). The bytes of the `credential`'s string encode the following (XDR) structure:

```
struct auth_unix {
        unsigned        stamp;
        string          machinename<255>;
        unsigned        uid;
        unsigned        gid;
        unsigned        gids<10>;
};
```

The `stamp` is an arbitrary ID that the caller machine may generate. The `machinename` is the name of the caller's machine

(like "krypton"). The uid is the caller's effective user ID. The gid is the caller's effective group ID. The gids is a counted array of groups that contain the caller as a member. The verifier accompanying the credentials should be of AUTH_NULL (defined above).

The value of the discriminate of the response verifier received in the reply message from the server may be AUTH_NULL or AUTH_SHORT(2). In the case of AUTH_SHORT, the bytes of the response verifier's string encode an auth_opaque structure. This new auth_opaque structure may now be passed to the server instead of the original AUTH_UNIX flavor credentials. The server keeps a cache that maps shorthand auth_opaque structures (passed back via a AUTH_SHORT style response verifier) to the original credentials of the caller. The caller can save network bandwidth and server cpu cycles by using the new credentials.

The server may flush the shorthand auth_opaque structure at any time. If this happens, the remote procedure call message will be rejected due to an authentication error. The reason for the failure will be AUTH_REJECTEDCRED. At this point, the caller may wish to try the original AUTH_UNIX style of credentials.

5.6 Record Marking Standard

When RPC messages are passed on top of a byte stream protocol (like TCP/IP), it is necessary, or at least desirable, to delimit one message from another in order to detect and possibly recover from user protocol errors. This is called record marking (RM). This RM/TCP/IP transport is used for passing RPC messages on TCP streams. One RPC message fits into one RM record.

A record is composed of one or more record fragments. A record fragment is a 4-byte header followed by 0 to 2^{31}-1 bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values – a boolean which indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment) and a 31-bit unsigned binary value which is the length in bytes of the fragment's data. The boolean value is the highest-order bit of the header; the length is the 31 low-order bits.

Note that this record specification is *not* in XDR standard form.

5.7 Port Mapper Program Protocol

The port mapper program maps RPC program and version numbers to UDP/IP or TCP/IP port numbers. This program makes dynamic binding of remote programs possible.

This is desirable because the range of reserved port numbers is very small and the number of potential remote programs is very large. By running only the port mapper on a reserved port, the port numbers of other remote programs can be ascertained by querying the port mapper.

5.7.1 The Port Mapper RPC Protocol

The protocol is specified by the XDR description language:

```
Port Mapper RPC Program Number: 100000
        Version Number: 1
        Supported Transports:
                UDP/IP on port 111
                RM/TCP/IP on port 111

/*
 * Handy transport protocol numbers
 */
#define IPPROTO_TCP         6
        /* protocol number used for rpc/rm/tcp/ip */
#define IPPROTO_UDP         17
        /* protocol number used for rpc/udp/ip */

/* Procedures */

/*
 * Convention: procedure zero of any protocol takes no parameters
 * and returns no results.
 */
0. PMAPPROC_NULL () returns ()

/*
 * Procedure 1, setting a mapping:
 * When a program first becomes available on a
 * machine, it registers itself with the port mapper program on
 * the same machine.  The program passes its program number
 * (prog), version number (vers), transport protocol number (prot),
 * and the port (port) on which it awaits service request.  The
 * procedure returns success whose value is TRUE if the
 * procedure successfully established the mapping and FALSE
 * otherwise.  The procedure will refuse to establish a mapping
 * if one already exists for the tuple [prog, vers, prot].
 */
1. PMAPPROC_SET (prog, vers, prot, port) returns (success)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned port;
        boolean success;
```

```
/*
 * Procedure 2, Unsetting a mapping:
 * When a program becomes unavailable, it should unregister
 * itself with the port mapper program on the same machine.
 * The parameters and results have meanings identical to those
 * of PMAPPROC_SET.
 */
2. PMAPPROC_UNSET (prog, vers, dummy1, dummy2) returns (success)
        unsigned prog;
        unsigned vers;
        unsigned dummy1;  /* this value is always ignored */
        unsigned dummy2;  /* this value is always ignored */
        boolean success;

/*
 * Procedure 3, looking-up a mapping:
 * Given a program number (prog), version number (vers) and
 * transport protocol number (prot), this procedure returns
 * the port number on which the program is awaiting call
 * requests.  A port value of zeros means that the program
 * has not been registered.
 */
3. PMAPPROC_GETPORT (prog, vers, prot, dummy) returns (port)
        unsigned prog;
        unsigned vers;
        unsigned prot;
        unsigned dummy;  /* this value is always ignored */
        unsigned port;  /* zero means the program is not */
                        /* registered */

/*
 * Procedure 4, dumping the mappings:
 * This procedure enumerates all entries in the port mapper's
 * database.  This procedure takes no parameters and returns
 * a "list" of [program, version, prot, port] values.
 */
4. PMAPPROC_DUMP () returns (maplist)
        struct maplist {
                union switch (boolean) {
                      FALSE: struct { /* void, end of list */ };
                      TRUE: struct {
                              unsigned prog;
                              unsigned vers;
                              unsigned prot;
                              unsigned port;
                              struct maplist the_rest;
                      };
                };
        } maplist;
```

```
/*
 * Procedure 5, indirect call routine:
 * The procedure allows a caller to call another remote
 * procedure on the same machine without knowing the remote
 * procedure's port number.  Its intended use is for
 * supporting broadcasts to arbitrary remote programs
 * via the well-known port mapper's port.  The parameters
 * prog, vers, proc, and the bytes of args are the program
 * number, version number, procedure number, and
 * parameters of the remote procedure.
 *
 * NB:
 * 1. This procedure only sends a response if the procedure was
 * successfully executed and is silent (No response) otherwise.
 * 2. The port mapper communicates with the remote program via
 * UDP/IP only.
 *
 * The procedure returns the port number of the remote program
 * and the bytes of results are the results of the remote
 * procedure.
 */
5. PMAPPROC_CALLIT (prog, vers, proc, args) returns (port, results)
        unsigned prog;
        unsigned vers;
        unsigned proc;
        string args<>;
        unsigned port;
        string results<>;
```

6. XDR PROTOCOL SPECIFICATION

6.1 Introduction

This section describes library routines that allow a C programmer to describe arbitrary data structures in a machine-independent fashion. The eXternal Data Representation (XDR) standard is the backbone of the Remote Procedure Call package, in the sense that data for remote procedure calls is transmitted using the standard. XDR library routines should be used to transmit data that is accessed (read or written) by more than one type of machine.

This section contains a description of XDR library routines, a guide to accessing currently available XDR streams, information on defining new streams and data types, and a formal definition of the XDR standard. XDR was designed to work across different languages, operating systems, and machine architectures. Most users (particularly RPC users) only need the information in sections 6.2 and 6.3. Programmers wishing to implement RPC and XDR on new machines will need the information in sections 6.4, 6.5, and 6.6. Advanced topics, not necessary for all implementations, are covered in section 6.7.

C programmers who want to use XDR routines should include in their programs the file `<rpc/rpc.h>`, which contains all of the necessary interfaces to the XDR system. Programs should be compiled as:

```
$ cc program.c -lrpc -linet
```

The compile flag `-lrpc` will request the inclusion of the RPC library `librpc.a`, and `-linet` includes the networking library.

6.2 Justification

Consider the following two programs, `writer`:

```
#include <stdio.h>

main()                          /* writer.c */
{
     long i;

     for (i = 0; i < 8; i++) {
          if (fwrite((char *)&i, sizeof(i), 1, stdout) != 1) {
               fprintf(stderr, "failed!\n");
               exit(1);
          }
     }
}
```

and `reader`:

```
#include <stdio.h>

main()                          /* reader.c */
{
      long i, j;

      for (j = 0; j < 8; j++) {
            if (fread((char *)&i, sizeof (i), 1, stdin) != 1) {
                  fprintf(stderr, "failed!\n");
                  exit(1);
            }
            printf("%ld ", i);
      }
      printf("\n");
}
```

The two programs appear to be portable, because

a. They pass `lint` checking.

b. They exhibit the same behavior when executed on two different hardware architectures, for example, a Sun and a VAX*.

Piping the output of the `writer` program to the `reader` program gives identical results on a Sun or a VAX.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%
---
vax% writer | reader
0 1 2 3 4 5 6 7
vax%
```

With the advent of local area networks came the concept of "network pipes" – a process produces data on one machine, and a second process consumes data on another machine. A network pipe can be constructed with `writer` and `reader`. Here are the results if the first produces data on a Sun and the second consumes data on a VAX:

```
sun% writer | rsh vax reader
0 16777216 33554432 50331648 67108864 83886080 100663296
117440512
sun%
```

Identical results can be obtained by executing `writer` on the VAX and `reader` on the Sun. These results occur because the byte ordering of long integers differs between the VAX and the Sun, even though word size is the same. Note that 16777216 is 2^{24} – when four bytes are reversed, the 1 winds up in the 24th bit.

Whenever data is shared by two or more machine types, there is a need for portable data. Programs can be made data-portable by replacing the `read()` and `write()` calls with calls to an XDR library routine `xdr_long()`, a filter that knows the standard representation of a long integer in its external form. Shown below are the revised versions of `writer`:

```
#include <stdio.h>
#include <rpc/rpc.h>          /* xdr is a sub-library of the rpc */
                              /* library */

main()                  /* writer.c */
{
        XDR xdrs;
        long i;

        xdrstdio_create(&xdrs, stdout, XDR_ENCODE);
        for (i = 0; i < 8; i++) {
                if (! xdr_long(&xdrs, &i)) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
        }
}
```

and `reader`:

```
#include <stdio.h>
#include <rpc/rpc.h>          /* xdr is a sub-library of the rpc */
                              /* library */

main()                  /* reader.c */
{
        XDR xdrs;
        long i, j;

        xdrstdio_create(&xdrs, stdin, XDR_DECODE);
        for (j = 0; j < 8; j++) {
                if (! xdr_long(&xdrs, &i)) {
                        fprintf(stderr, "failed!\n");
                        exit(1);
                }
                printf("%ld ", i);
        }
        printf("\n");
}
```

The new programs were executed on a Sun, on a VAX, and from a Sun to a VAX; the results are shown below.

```
sun% writer | reader
0 1 2 3 4 5 6 7
sun%
---
vax% writer | reader
0 1 2 3 4 5 6 7
vax%
---
sun% writer | rsh vax reader
0 1 2 3 4 5 6 7
sun%
```

Dealing with integers is only a small part of portable data. Arbitrary data structures present portability problems, particularly with respect to alignment and pointers. Alignment on word boundaries may cause the size of a structure to vary from machine to machine. Pointers are convenient to use, but they have no meaning outside the machine where they are defined.

The XDR library package solves data portability problems. It allows users to write and read arbitrary C constructs in a consistent, specified, well documented manner. Thus, it makes sense to use the library even when the data is not shared among machines on a network.

The XDR library has filter routines for many subjects, including strings (null-terminated arrays of bytes), structures, unions, and arrays, to name a few. Using more primitive routines, users can write their own specific XDR routines to describe arbitrary data structures, including elements of arrays, arms of unions, or objects pointed at from other structures. The structures themselves may contain arrays of arbitrary elements, or pointers to other structures.

The two programs are examined more closely below.

A family of XDR stream creation routines exists, in which each member treats the stream of bits differently. In the example given, data is manipulated using standard I/O routines, so `xdrstdio_create()` is used. The parameters to XDR stream creation routines vary according to their function. In the example, `xdrstdio_create()` takes a pointer to an XDR structure that it initializes, a pointer to a `FILE` that the input or output is performed on, and the operation. The operation may be `XDR_ENCODE` for serializing in the `writer` program or `XDR_DECODE` for deserializing in the `reader` program.

Note that RPC clients never need to create XDR streams; the RPC system itself creates these streams, which are then passed to the clients.

The `xdr_long()` primitive is characteristic of most XDR library primitives and all client XDR routines. First, the routine returns `FALSE` (0) if it fails and `TRUE` (1) if it succeeds. Second, for each data type, *xxx*, there is an associated XDR routine of the form:

```
xdr_xxx(xdrs, fp)
        XDR *xdrs;
        xxx *fp;
{
}
```

In this case, *xxx* is long, and the corresponding XDR routine is a primitive, `xdr_long`. The client could also define an arbitrary structure *xxx*, in which case the client would also supply the routine `xdr_xxx`, describing each field by calling XDR routines of the appropriate type. In all cases the first parameter, `xdrs`, can be treated as an opaque handle and passed to the primitive routines.

XDR routines are direction independent; that is, the same routines are called to serialize or deserialize data. This feature is critical to software engineering of portable data. The intention is to call the same routine for either operation; this almost guarantees that serialized data can also be deserialized. One routine is used by both producer and consumer of networked data. This is implemented by always passing the address of an object rather than the object itself, only in the case of deserialization is the object modified. This feature is not shown in the example, but its value becomes obvious when nontrivial data structures are passed among machines. If needed, the direction of the XDR operation can be obtained. See section 6.7 for details.

Consider a slightly more complicated example. Assume that a person's gross assets and liabilities are to be exchanged among processes; also assume that these values are important enough to warrant their own data type:

```
struct gnumbers {
        long g_assets;
        long g_liabilities;
};
```

The corresponding XDR routine describing this structure would be:

```
bool_t                  /* TRUE is success, FALSE is failure */
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        if (xdr_long(xdrs, &gp->g_assets) &&
            xdr_long(xdrs, &gp->g_liabilities))
                return (TRUE);
        return (FALSE);
}
```

Note that the parameter `xdrs` is never inspected or modified; it is only passed on to the subcomponent routines. It is imperative to inspect the return value of each XDR routine call, and to give up immediately and return `FALSE` if the subroutine fails.

This example also shows that the type `bool_t` is declared as an integer whose only values are `TRUE` (1) and `FALSE` (0). This documentation uses the following definitions:

```
#define bool_t   int
#define TRUE       1
#define FALSE      0

#define enum_t int  /* enum_t's are used for generic enum's */
```

Using these conventions, `xdr_gnumbers()` can be rewritten as follows:

```
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        return (xdr_long(xdrs, &gp->g_assets) &&
                xdr_long(xdrs, &gp->g_liabilities));
}
```

This documentation uses both coding styles.

6.3 XDR Library Primitives

This section gives a synopsis of each XDR primitive. It starts with basic data types and moves on to constructed data types. Finally, XDR utilities are discussed. The interface to these primitives and utilities is defined in the `include` file `<rpc/xdr.h>`, which is automatically included by `<rpc/rpc.h>`.

6.3.1 Number Filters

The XDR library provides primitives that translate between C numbers and their corresponding external representations. The primitives cover the set of numbers in:

```
[signed, unsigned] * [short, int, long]
```

Specifically, the six primitives are:

```
bool_t xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;

bool_t xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;

bool_t xdr_long(xdrs, lip)
        XDR *xdrs;
        long *lip;

bool_t xdr_u_long(xdrs, lup)
        XDR *xdrs;
        ulong *lup;

bool_t xdr_short(xdrs, sip)
        XDR *xdrs;
        short *sip;

bool_t xdr_u_short(xdrs, sup)
        XDR *xdrs;
        ushort *sup;
```

The first parameter, **xdrs**, is an XDR stream handle. The second parameter is the address of the number that provides data to the stream or receives data from it. All routines return **TRUE** if they complete successfully and **FALSE** otherwise.

6.3.2 Floating Point Filters

The XDR library also provides primitive routines for C's floating point types:

```
bool_t xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;

bool_t xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

The first parameter, **xdrs**, is an XDR stream handle. The second parameter is the address of the floating point number that provides data to the stream or receives data from it. All routines return **TRUE** if they complete successfully and **FALSE** otherwise.

Note that since the numbers are represented in IEEE floating point format, routines may fail when decoding a valid IEEE representation into a machine-specific representation or vice versa.

6.3.3 Enumeration Filters

The XDR library provides a primitive for generic enumerations. The primitive assumes that a C `enum` has the same representation inside the machine as a C integer. The boolean type is an important instance of the `enum`. The external representation of a boolean is always 1 (`TRUE`) or 0 (`FALSE`):

```
#define bool_t    int
#define FALSE       0
#define TRUE        1

#define enum_t int

bool_t xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;

bool_t xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

The second parameters `ep` and `bp` are addresses of the associated type that provides data to, or receives data from, the stream `xdrs`. The routines return `TRUE` if they complete successfully and `FALSE` otherwise.

6.3.4 No Data

Occasionally, an XDR routine must be supplied to the RPC system, even when no data is passed or required. The library provides such a routine:

```
bool_t xdr_void();  /* always returns TRUE */
```

6.3.5 Constructed Data Type Filters

Constructed or compound data type primitives require more parameters and perform more complicated functions than the primitives discussed above. This section includes primitives for strings, arrays, unions, and pointers to structures.

Constructed data type primitives may use memory management. In many cases, memory is allocated when deserializing data with `XDR_DECODE`. Therefore, the XDR package must provide means to deallocate memory. This is done by an XDR operation, `XDR_FREE`. To review; the three XDR directional operations are `XDR_ENCODE`, `XDR_DECODE`, and `XDR_FREE`.

6.3.5.1 Strings. In C, a string is defined as a sequence of bytes terminated by a null byte, which is not considered when calculating string length. However, when a string is passed or manipulated, a pointer to it is employed. Therefore, the XDR library defines a string to be a `char *`, and not a sequence of characters. The external representation of a string is drastically different from its internal representation. Externally strings are represented as sequences of ASCII characters, while internally they are represented with character pointers. Conversion between the two representations is accomplished with the routine `xdr_string()`:

```
bool_t xdr_string(xdrs, sp, maxlength)
        XDR *xdrs;
        char **sp;
        uint maxlength;
```

The first parameter `xdrs` is the XDR stream handle. The second parameter `sp` is a pointer to a string (type `char **`). The third parameter `maxlength` specifies the maximum number of bytes allowed during encoding or decoding; its value is usually specified by a protocol. For example, a protocol specification may say that a file name may be no longer than 255 characters. The routine returns `FALSE` if the number of characters exceeds `maxlength` and `TRUE` if it does not.

The behavior of `xdr_string()` is similar to the behavior of other routines discussed in this section. The direction `XDR_ENCODE` is easiest to understand. The parameter `sp` points to a string of a certain length; if it does not exceed `maxlength`, the bytes are serialized.

The effect of deserializing a string is subtle. First the length of the incoming string is determined; it must not exceed `maxlength`. Next `sp` is dereferenced; if the value is `NULL`, a string of the appropriate length is allocated and `*sp` is set to this string. If the original value of `*sp` is non-NULL, the XDR package assumes that a target area, which can hold strings no longer than `maxlength`, has been allocated. In either case, the string is decoded into the target area. The routine then appends a null character to the string.

In the `XDR_FREE` operation, the string is obtained by dereferencing `sp`. If the string is not `NULL`, it is freed and `*sp` is set to `NULL`. In this operation, `xdr_string` ignores the `maxlength` parameter.

6.3.5.2 Byte Arrays. Often variable-length arrays of bytes are preferable to strings. Byte arrays differ from strings in the following three ways:

1. The length of the array (the byte count) is explicitly located in an unsigned integer.
2. The byte sequence is not terminated by a null character.
3. The external representation of the bytes is the same as their internal representation.

The primitive `xdr_bytes()` converts between the internal and external representations of byte arrays:

```
bool_t xdr_bytes(xdrs, bpp, lp, maxlength)
        XDR *xdrs;
        char **bpp;
        uint *lp;
        uint maxlength;
```

The usage of the first, second, and fourth parameters are identical to the first, second, and third parameters of `xdr_string()`, respectively. The length of the byte area is obtained by dereferencing `lp` when serializing; `*lp` is set to the byte length when deserializing.

6.3.5.3 Arrays. The XDR library package provides a primitive for handling arrays of arbitrary elements. The `xdr_bytes()` routine treats a subset of generic arrays in which the size of array elements is known to be 1 and the external description of each element is built-in. The generic array primitive `xdr_array()` requires parameters identical to those of `xdr_bytes()` plus two more: the size of array elements and an XDR routine to handle each of the elements. This routine is called to encode or decode each element of the array:

```
bool_t xdr_array(xdrs, ap, lp, maxlength, elementsize, xdr_element)
        XDR *xdrs;
        char **ap;
        uint *lp;
        uint maxlength;
        uint elementsize;
        bool_t (*xdr_element)();
```

The parameter `ap` is the address of the pointer to the array. If `*ap` is `NULL` when the array is being deserialized, XDR allocates an array of the appropriate size and sets `*ap` to that array. The element count of the array is obtained from `*lp` when the array is serialized; `*lp` is set to the array length when the array is deserialized. The parameter `maxlength` is the maximum number of

elements that the array is allowed to have; `elementsize` is the byte size of each element of the array (the C function `sizeof()` can be used to obtain this value). The routine `xdr_element` is called to serialize, deserialize, or free each element of the array.

Examples

Before more constructed data types are defined, review the following three examples (Examples A, B, and C).

Example A

A user on a networked machine can be identified by

a. The machine name, such as `krypton`; see *gethostname*(3).

b. The user's UID; see *getuid*(2).

c. The user's GID or set of GIDs; see *getgid*(2).

A structure with this information and its associated XDR routine could be coded as follows:

```
struct netuser {
        char    *nu_machinename;
        int     nu_uid;
        uint    nu_glen;
        int     *nu_gids;
};
#define NLEN 255 /* machine names must be shorter than 256 */
                 /* chars */
#define NGRPS 20 /* user can't be a member of more than 20 */
                 /* groups */
bool_t
xdr_netuser(xdrs, nup)
        XDR *xdrs;
        struct netuser *nup;
{
        return (xdr_string(xdrs, &nup->nu_machinename, NLEN) &&
            xdr_int(xdrs, &nup->nu_uid) &&
            xdr_array(xdrs, &nup->nu_gids, &nup->nu_glen, NGRPS,
                sizeof (int), xdr_int));
}
```

Example B

A party of network users could be implemented as an array of `netuser` structures. The declaration and its associated XDR routines are as follows:

```
struct party {
        uint p_len;
        struct netuser *p_nusers;
};
#define PLEN 500 /* max number of users in a party */

bool_t
xdr_party(xdrs, pp)
        XDR *xdrs;
        struct party *pp;
{
        return (xdr_array(xdrs, &pp->p_nusers, &pp->p_len, PLEN,
            sizeof (struct netuser), xdr_netuser));
}
```

Example C

The well-known parameters to `main()`, `argc`, and `argv` can be combined into a structure. An array of these structures can make up a history of commands. The declarations and XDR routines might look like:

```
struct cmd {
        uint c_argc;
        char **c_argv;
};
#define ALEN 1000  /* args can be no longer than 1000 chars */
#define NARGC 100  /* commands may have no more than 100 args */

struct history {
        uint h_len;
        struct cmd *h_cmds;
};
#define NCMDS 75  /* history is no more than 75 commands */

bool_t
xdr_wrap_string(xdrs, sp)
        XDR *xdrs;
        char **sp;
{
        return (xdr_string(xdrs, sp, ALEN));
}

bool_t
xdr_cmd(xdrs, cp)
        XDR *xdrs;
        struct cmd *cp;
{
        return (xdr_array(xdrs, &cp->c_argv, &cp->c_argc, NARGC,
            sizeof (char *), xdr_wrap_string));
}

bool_t
xdr_history(xdrs, hp)
        XDR *xdrs;
        struct history *hp;
{
        return (xdr_array(xdrs, &hp->h_cmds, &hp->h_len, NCMDS,
            sizeof (struct cmd), xdr_cmd));
}
```

Note that the routine `xdr_wrap_string()` is needed to package the `xdr_string()` routine because the implementation of `xdr_array()` only passes two parameters to the array element description routine; `xdr_wrap_string()` supplies the third parameter to `xdr_string()`.

By now the recursive nature of the XDR library should be obvious. The following sections describe more constructed data types.

6.3.5.4 Opaque Data. In some protocols, handles are passed from a server to client. The client passes the handle back to the server at some later time. Handles are never inspected by clients; they are obtained and submitted. That is to say, handles are opaque. The primitive `xdr_opaque()` is used for describing fixed-sized, opaque bytes:

```
bool_t xdr_opaque(xdrs, p, len)
        XDR *xdrs;
        char *p;
        uint len;
```

The parameter `p` is the location of the bytes; `len` is the number of bytes in the opaque object. By definition, the actual data contained in the opaque object are not machine portable.

6.3.5.5 Fixed-Length Arrays. The XDR library does not provide a primitive for fixed-length arrays (the primitive `xdr_array()` is for variable-length arrays). Example A could be rewritten to use fixed-length arrays in the following fashion:

```
#define NLEN 255  /* machine names must be shorter than 256 */
                  /* chars */
#define NGRPS 20  /* user cannot be a member of more than 20 */
                  /* groups */

struct netuser {
        char *nu_machinename;
        int nu_uid;
        int nu_gids[NGRPS];
};
```

```
bool_t
xdr_netuser(xdrs, nup)
        XDR *xdrs;
        struct netuser *nup;
{
        int i;

        if (! xdr_string(xdrs, &nup->nu_machinename, NLEN))
                return (FALSE);
        if (! xdr_int(xdrs, &nup->nu_uid))
                return (FALSE);
        for (i = 0; i < NGRPS; i++) {
                if (! xdr_int(xdrs, &nup->nu_gids[i]))
                        return (FALSE);
        }
        return (TRUE);
}
```

6.3.5.6 Discriminated Unions. The XDR library supports discriminated unions. A discriminated union is a C union and an `enum_t` value that selects an "arm" of the union:

```
struct xdr_discrim {
        enum_t value;
        bool_t (*proc)();
};

bool_t xdr_union(xdrs, dscmp, unp, arms, defaultarm)
        XDR *xdrs;
        enum_t *dscmp;
        char *unp;
        struct xdr_discrim *arms;
        bool_t (*defaultarm)();  /* may equal NULL */
```

First the routine translates the discriminant of the union located at `*dscmp`. The discriminant is always an `enum_t`. Next, the union located at `*unp` is translated. The parameter `arms` is a pointer to an array of `xdr_discrim` structures. Each structure contains an order pair of `[value, proc]`. If the union's discriminant is equal to the associated `value`, the `proc` is called to translate the union. The end of the `xdr_discrim` structure array is denoted by a routine of value `NULL` (0). If the discriminant is not found in the `arms` array, the `defaultarm` procedure is called if it is non-`NULL`; otherwise the routine returns `FALSE`.

Example D

Suppose the type of a union may be integer, character pointer (a string), or a `gnumbers` structure. Also, assume the union and its current type are declared in a structure. The declaration is:

```
enum utype { INTEGER=1, STRING=2, GNUMBERS=3 };

struct u_tag {
     enum utype utype;      /* this is the union's discriminant */
     union {
             int ival;
             char *pval;
             struct gnumbers gn;
     } uval;
};
```

The following constructs and XDR procedure (de)serialize the discriminated union:

```
struct xdr_discrim u_tag_arms[4] = {
        { INTEGER, xdr_int },
        { GNUMBERS, xdr_gnumbers },
        { STRING, xdr_wrap_string },
        { __dontcare__, NULL }
        /* always terminate arms with a NULL xdr_proc */
}

bool_t
xdr_u_tag(xdrs, utp)
        XDR *xdrs;
        struct u_tag *utp;
{
        return (xdr_union(xdrs, &utp->utype, &utp->uval, u_tag_arms,
            NULL));
}
```

The routine `xdr_gnumbers()` was presented in section 6.2; `xdr_wrap_string()` was presented in Example C. The default arm parameter to `xdr_union()` (the last parameter) is `NULL` in this example. Therefore, the value of the union's discriminant may legally take on only values listed in the `u_tag_arms` array. This example also demonstrates that the elements of the arm's array do not need to be sorted.

It should be noted that the values of the discriminant may be sparse, though in this example they are not. It is always good practice to assign explicit integer values to each element of the discriminant's type. This practice both documents the external representation of the discriminant and guarantees that different C compilers emit identical discriminant values.

6.3.5.7 Pointers. In C it is often convenient to put pointers to another structure within a structure. The primitive `xdr_reference()` makes it easy to serialize, deserialize, and free these referenced structures.

```
bool_t xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        uint ssize;
        bool_t (*proc)();
```

Parameter `pp` is the address of the pointer to the structure, parameter `ssize` is the size in bytes of the structure (the C function `sizeof()` may be used to obtain this value), and `proc` is the XDR routine that describes the structure. When decoding data, storage is allocated if `*pp` is `NULL`.

There is no need for a primitive `xdr_struct()` to describe structures within structures, because pointers are always sufficient.

Note that `xdr_reference()` and `xdr_array()` are *not* interchangeable external representations of data.

Example E

Suppose there is a structure containing a person's name and a pointer to a `gnumbers` structure containing the person's gross assets and liabilities. The construct is:

```
struct pgn {
        char *name;
        struct gnumbers *gnp;
};
```

The corresponding XDR routine for this structure is:

```
bool_t
xdr_pgn(xdrs, pp)
        XDR *xdrs;
        struct pgn *pp;
{
        if (xdr_string(xdrs, &pp->name, NLEN) &&
            xdr_reference(xdrs, &pp->gnp, sizeof(struct gnumbers),
            xdr_gnumbers))
                return (TRUE);
        return (FALSE);
}
```

6.3.5.8 Pointer Semantics and XDR. In many applications, C programmers attach double meaning to the values of a pointer. Typically the value `NULL` (or zero) means data is not needed, yet some application-specific interpretation applies. In essence, the C programmer is encoding a discriminated union efficiently by overloading the interpretation of the value of a pointer. For instance, in Example E a `NULL` pointer value for `gnp` could indicate that the person's assets and liabilities are unknown. That is, the pointer value encodes two things – whether or not the data is known and, if it is known, where it is located in memory. Linked lists are an extreme example of the use of application-specific pointer interpretation.

The primitive `xdr_reference()` cannot and does not attach any special meaning to a `NULL`-value pointer during serialization. That is, passing an address of a pointer whose value is `NULL` to `xdr_reference()` when serialing data will most likely cause a memory fault and, on the INTERACTIVE UNIX Operating System, a core dump for debugging.

It is the explicit responsibility of the programmer to expand non-referencable pointers into their specific semantics. This usually involves describing data with a two-armed discriminated union. One arm is used when the pointer is valid; the other is used when the pointer is invalid (`NULL`). Section 6.7 has an example (linked lists encoding) that deals with invalid pointer interpretation.

6.3.6 Non-Filter Primitives

XDR streams can be manipulated with the primitives discussed in this section:

```
uint xdr_getpos(xdrs)
        XDR *xdrs;

bool_t xdr_setpos(xdrs, pos)
        XDR *xdrs;
        uint pos;

xdr_destroy(xdrs)
        XDR *xdrs;
```

The routine `xdr_getpos()` returns an unsigned integer that describes the current position in the data stream.

WARNING: In some XDR streams, the returned value of `xdr_getpos()` is meaningless; the routine returns a -1 in this case (though -1 should be a legitimate value).

The routine `xdr_setpos()` sets a stream position to `pos`.

WARNING: In some XDR streams, setting a position is impossible; in such cases, `xdr_setpos()` will return `FALSE`. This routine will also fail if the requested position is out-of-bounds. The definition of bounds varies from stream to stream.

The `xdr_destroy()` primitive destroys the XDR stream. Use of the stream after calling this routine is undefined.

6.3.7 XDR Operation Directions

At times it may be wished to optimize XDR routines by taking advantage of the direction of the operation (`XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`). The value `xdrs->x_op` always contains the direction of the XDR operation. Programmers are not encouraged to take advantage of this information. Therefore, no example is presented here. However, an example in section 6.7 demonstrates the usefulness of the `xdrs\->x_op` field.

6.4 XDR Stream Access

An XDR stream is obtained by calling the appropriate creation routine. These creation routines take arguments that are tailored to the specific properties of the stream.

Streams currently exist for (de)serialization of data to or from standard I/O `FILE` streams, TCP/IP connections and INTERACTIVE UNIX System files, and memory. Section 6.5 documents the XDR object and how to make new XDR streams when they are required.

6.4.1 Standard I/O Streams

XDR streams can be interfaced to standard I/O using the `xdrstdio_create()` routine as follows:

```
#include <stdio.h>
#include <rpc/rpc.h>  /* xdr streams are a part of the rpc */
                      /* library */

void
xdrstdio_create(xdrs, fp, x_op)
        XDR *xdrs;
        FILE *fp;
        enum xdr_op x_op;
```

The routine `xdrstdio_create()` initializes an XDR stream pointed to by `xdrs`. The XDR stream interfaces to the standard I/O library. Parameter `fp` is an open file, and `x_op` is an XDR direction.

6.4.2 Memory Streams

Memory streams allow the streaming of data into or out of a specified area of memory.

```
#include <rpc/rpc.h>

void
xdrmem_create(xdrs, addr, len, x_op)
        XDR *xdrs;
        char *addr;
        uint len;
        enum xdr_op x_op;
```

The routine `xdrmem_create()` initializes an XDR stream in local memory. The memory is pointed to by parameter `addr`; parameter `len` is the length in bytes of the memory. The parameters `xdrs` and `x_op` are identical to the corresponding parameters of `xdrstdio_create()`. Currently the UDP/IP implementation of RPC uses `xdrmem_create()`. Complete call or result messages are built in memory before calling the `sendto()` system routine.

6.4.3 Record (TCP/IP) Streams

A record stream is an XDR stream built on top of a record marking standard that is built on top of the INTERACTIVE UNIX System file or network connection interface.

```
#include <rpc/rpc.h>  /* xdr streams are a part of the rpc */
                      /* library /*
xdrrec_create(xdrs, sendsize, recvsize, iohandle, readproc,
writeproc)
        XDR *xdrs;
        uint sendsize, recvsize;
        char *iohandle;
        int (*readproc)(), (*writeproc)();
```

The routine `xdrrec_create()` provides an XDR stream interface that allows for a bidirectional, arbitrarily long sequence of records. The contents of the records are meant to be data in XDR form. The stream's primary use is for interfacing RPC to TCP connections. However, it can be used to stream data into or out of normal UNIX System files.

The parameter `xdrs` is similar to the corresponding parameter described above. The stream does its own data buffering similar to that of standard I/O. The parameters `sendsize` and `recvsize` determine the size in bytes of the output and input buffers, respectively; if their values are zero (`0`), then predetermined defaults are used. When a buffer needs to be filled or flushed, the routines `readproc` or `writeproc`, respectively, are called.

The usage and behavior of these routines are similar to the UNIX Operating System system calls `read()` and `write()`; however, the first parameter to each of these routines is the opaque parameter `iohandle`. The other two parameters (`buf` and `nbytes`) and the results (byte count) are identical to the system routines. If *xxx* is `readproc` or `writeproc`, then it has the following form:

```
/* returns the actual number of bytes transferred.
 * -1 is an error
 */
int
xxx(iohandle, buf, len)
        char *iohandle;
        char *buf;
        int nbytes;
```

The XDR stream provides the means for delimiting records in the byte stream. The implementation details of delimiting records in a stream are discussed in section 6.8. The primitives that are specific to record streams are as follows:

```
bool_t
xdrrec_endofrecord(xdrs, flushnow)
        XDR *xdrs;
        bool_t flushnow;

bool_t
xdrrec_skiprecord(xdrs)
        XDR *xdrs;

bool_t
xdrrec_eof(xdrs)
        XDR *xdrs;
```

The routine `xdrrec_endofrecord()` causes the current outgoing data to be marked as a record. If the parameter `flushnow` is `TRUE`, then the stream's `writeproc()` will be called; otherwise, `writeproc()` will be called when the output buffer has been filled.

The routine `xdrrec_skiprecord()` causes an input stream's position to be moved past the current record boundary and onto the beginning of the next record in the stream.

If there is no more data in the stream's input buffer, then the routine `xdrrec_eof()` returns `TRUE`. This does not imply that there is no more data in the underlying file descriptor.

6.5 XDR Stream Implementation

This section provides the abstract data types needed to implement new instances of XDR streams.

6.5.1 The XDR Object

The following structure defines the interface to an XDR stream:

```
enum xdr_op { XDR_ENCODE = 0, XDR_DECODE = 1, XDR_FREE = 2 };

typedef struct {
    enum    xdr_op x_op; /* operation; fast additional param */
    struct  xdr_ops {
       bool_t  (*x_getlong)();     /* get a long from */
                                   /* underlying stream */
       bool_t  (*x_putlong)();     /* put a long to " */
       bool_t  (*x_getbytes)();    /* get some bytes from " */
       bool_t  (*x_putbytes)();    /* put some bytes to " */
       uint    (*x_getpostn)();    /* returns byte offset from */
                                   /* beginning */
       bool_t  (*x_setpostn)();    /* repositions position in */
                                   /* stream */
       caddr_t (*x_inline)();      /* buf quick ptr to buffered */
                                   /* data */
       VOID    (*x_destroy)();     /* free privates of this */
                                   /* xdr_stream */
    }       *x_ops;
    caddr_t x_public;     /* users' data */
    caddr_t x_private;    /* pointer to private data */
    caddr_t x_base;       /* private used for position info */
    int     x_handy;      /* extra private word */
} XDR;
```

The `x_op` field is the current operation being performed on the stream. This field is important to the XDR primitives, but it should not affect the implementation of a stream. That is, the implementation of a stream should not depend on this value. The fields `x_private`, `x_base`, and `x_handy` are private to the particular stream's implementation. The field `x_public` is for the XDR client and should never be used by the XDR stream implementations or the XDR primitives.

Macros for accessing operations `x_getpostn()`, `x_setpostn()`, and `x_destroy()` were defined in section 6.3.6. The operation `x_inline()` takes two parameters – an `XDR *` and an unsigned integer, which is a byte count. The routine returns a pointer to a piece of the stream's internal buffer. The caller can then use the buffer segment for any purpose. From the point of view of the stream, the bytes in the buffer segment have been consumed or put. The routine may return `NULL` if it cannot return a buffer segment of the requested size. (The `x_inline` routine is for cycle squeezers. Use of the resulting buffer is not data-portable. Users are encouraged not to use this feature.)

The operations `x_getbytes()` and `x_putbytes()` blindly get and put sequences of bytes from or to the underlying stream; they return `TRUE` if they are successful, and `FALSE` otherwise.

The routines have identical parameters (replace *xxx*):

```
bool_t
xxxbytes(xdrs, buf, bytecount)
        XDR *xdrs;
        char *buf;
        uint bytecount;
```

The operations `x_getlong()` and `x_putlong()` receive and put long numbers from and to the data stream. It is the responsibility of these routines to translate the numbers between the machine representation and the (standard) external representation. The UNIX System networking primitives `htonl()` and `ntohl()` can be helpful in accomplishing this. Section 6.6 defines the standard representation of numbers. The higher-level XDR implementation assumes that signed and unsigned long integers contain the same number of bits, and that non-negative integers have the same bit representations as unsigned integers. The routines return `TRUE` if they succeed, and `FALSE` otherwise. They have identical parameters:

```
bool_t
xxxlong(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

Implementors of new XDR streams must make an XDR structure (with new operation routines) available to clients using some kind of create routine.

6.6 XDR Standard

This section defines the external data representation standard. The standard is independent of languages, operating systems, and hardware architectures. Once data is shared among machines, it should not matter that the data was produced on a Sun, but is consumed by a VAX (or vice versa). Similarly the choice of operating systems should have no influence on how the data is represented externally. For programming languages, data produced by a C program should be readable by a FORTRAN or Pascal program.

The external data representation standard depends on the assumption that bytes (or octets) are portable. A byte is defined to be eight bits of data. It is assumed that hardware that encodes bytes onto various media will preserve the bytes' meanings across hardware boundaries. For example, the Ethernet standard suggests that bytes be encoded "little endian" style. Both Sun and VAX hardware implementations adhere to the standard.

The XDR standard also suggests a language used to describe data. The language is a modified C; it is a data description language, not a programming language. (The Xerox* Courier Standard uses modified Mesa as its data description language.)

6.6.1 Basic Block Size

The representation of all items requires a multiple of four bytes (or 32 bits) of data. The bytes are numbered 0 through $n-1$, where (n mod 4)=0. The bytes are read or written to some byte stream such that byte m always precedes byte $m+1$.

6.6.2 Integer

An XDR signed integer is a 32-bit datum that encodes an integer in the range [-2147483648, 2147483647]. The integer is represented in two's complement notation. The most and least significant bytes are 0 and 3, respectively. The data description of integers is `integer`.

6.6.3 Unsigned Integer

An XDR unsigned integer is a 32-bit datum that encodes a nonnegative integer in the range [0, 4294967295]. It is represented by an unsigned binary number whose most and least significant bytes are 0 and 3, respectively. The data description of unsigned integers is `unsigned`.

6.6.4 Enumerations

Enumerations have the same representation as integers. Enumerations are useful for describing subsets of the integers. The data description of enumerated data is as follows:

```
typedef enum { name = value, .... } type-name;
```

For example, the three colors red, yellow, and blue could be described by an enumerated type:

```
typedef enum { RED = 2, YELLOW = 3, BLUE = 5 } colors;
```

6.6.5 Booleans

Booleans are important enough and occur frequently enough to warrant their own explicit type in the standard. Boolean is an enumeration with the following form:

```
typedef enum { FALSE = 0, TRUE = 1 } boolean;
```

6.6.6 Hyper Integer and Hyper Unsigned

The standard also defines 64-bit (8-byte) numbers called `hyper integer` and `hyper unsigned`. Their representations are the obvious extensions of the integer and unsigned defined above. The most and least significant bytes are 0 and 7, respectively.

6.6.7 Floating Point and Double Precision

The standard defines the encoding for the floating point data types `float` (32 bits or 4 bytes) and `double` (64 bits or 8 bytes). The encoding used is the IEEE standard for normalized single- and double-precision floating point numbers. See the IEEE floating point standard for more information. The standard encodes the following three fields, which describe the floating point number:

- `S` The sign of the number. Values 0 and 1 represent positive and negative, respectively.
- `E` The exponent of the number, base 2. Floats devote 8 bits to this field, while doubles devote 11 bits. The exponents for float and double are biased by 127 and 1023, respectively.
- `F` The fractional part of the number's mantissa, base 2. Floats devote 23 bits to this field, while doubles devote 52 bits.

Therefore, the floating point number is described by:

$$(-1)^{S} \times 2^{E-Bias} * 1.F$$

Just as the most and least significant bytes of a number are 0 and 3, the most and least significant bits of a single-precision floating point number are 0 and 31. The beginning bit (and most significant bit) offsets of `S`, `E`, and `F` are 0, 1, and 9, respectively.

Doubles have the analogous extensions. The beginning bit (and most significant bit) offsets of `S`, `E`, and `F` are 0, 1, and 12, respectively.

The IEEE specification should be consulted concerning the encoding for signed zero, signed infinity (overflow), and denormalized numbers (underflow). Under IEEE specifications, the "NaN" (not a number) is system dependent and should not be used.

6.6.8 Opaque Data

At times fixed-length uninterpreted data needs to be passed among machines. This data is called `opaque` and is described as:

```
typedef opaque type-name[n];
opaque name[n];
```

where `n` is the (static) number of bytes necessary to contain the opaque data. If `n` is not a multiple of four, then the `n` bytes are followed by enough (up to 3) zero-valued bytes to make the total byte count of the opaque object a multiple of four.

6.6.9 Counted Byte Strings

The standard defines a string of *n* (numbered 0 through *n*-1) bytes to be the number *n* encoded as `unsigned`, and followed by the *n* bytes of the string. If *n* is not a multiple of four, then the *n* bytes are followed by enough (up to 3) zero-valued bytes to make the total byte count a multiple of four. The data description of strings is as follows:

```
typedef string type-name<N>;
typedef string type-name<>;
string name<N>;
string name<>;
```

Note that the data description language uses angle brackets (< and >) to denote anything that is of variable length (as opposed to square brackets to denote fixed-length sequences of data).

The constant `N` denotes an upper bound of the number of bytes that a string may contain. If `N` is not specified, it is assumed to be $2^{32}-1$, the maximum length. The constant `N` would normally be found in a protocol specification. For example, a filing protocol may state that a file name can be no longer than 14 bytes, such as:

```
string filename<14>;
```

The XDR specification does not say what the individual bytes of a string represent; this important information is left to higher-level specifications. A reasonable default is to assume that the bytes encode ASCII characters.

6.6.10 Fixed Arrays

The data description for fixed-length arrays of homogeneous elements is as follows:

```
typedef elementtype type-name[n];
elementtype name[n];
```

Fixed-size arrays of elements numbered 0 through $n-1$ are encoded by individually encoding the elements of the array in their natural order, 0 through $n-1$.

6.6.11 Counted Arrays

Counted arrays provide the ability to encode variable-length arrays of homogeneous elements. The array is encoded as the element count n (an unsigned integer), followed by the encoding of each of the array's elements, starting with element 0 and progressing through element $n-1$. The data description for counted arrays is similar to that of counted strings:

```
typedef elementtype type-name<N>;
typedef elementtype type-name<>;
elementtype name<N>;
elementtype name<>;
```

Again, the constant `N` specifies the maximum acceptable element count of an array; if `N` is not specified, it is assumed to be $2^{32}-1$.

6.6.12 Structures

The data description for structures is very similar to that of standard C:

```
typedef struct {
        component-type component-name;
        ...
} type-name;
```

The components of the structure are encoded in the order of their declaration in the structure.

6.6.13 Discriminated Unions

A discriminated union is a type composed of a discriminant followed by a type selected from a set of prearranged types according to the value of the discriminant. The type of the discriminant is always an enumeration. The component types are called "arms" of the union. The discriminated union is encoded as its discriminant followed by the encoding of the implied arm. The data description for discriminated unions is as follows:

```
typedef union switch (discriminant-type) {
        discriminant-value: arm-type;
        ...
        default: default-arm-type;
} type-name;
```

The default arm is optional. If it is not specified, then a valid encoding of the union cannot take on unspecified discriminant values. Most specifications neither need nor use default arms.

6.6.14 Missing Specifications

The standard lacks representations for bit fields and bitmaps, since the standard is based on bytes. This is not to say that no specification should be attempted.

6.6.15 Library Primitive/XDR Standard Cross Reference

The following table describes the association between the C library primitives discussed in section 6.3 and the standard data types defined in this section:

C Primitive	XDR Type	Sections
xdr_int xdr_long xdr_short	integer	6.3.1 6.6.2
xdr_u_int xdr_u_long xdr_u_short	unsigned	6.3.1 6.6.3
-	hyper integer hyper unsigned	6.6.6
xdr_float	float	6.3.2 6.6.7
xdr_double	double	6.3.2 6.6.7
xdr_enum	enum_t	6.3.3 6.6.4
xdr_bool	bool_t	6.3.3 6.6.5
xdr_string xdr_bytes	string	6.3.5.1 6.3.5.2 6.6.9
xdr_array	(variable arrays)	6.3.5.3 6.6.11
-	(fixed arrays)	6.3.5.5 6.6.10
xdr_opaque	opaque	6.3.5.4 6.6.8
xdr_union	union	6.3.5.6 6.6.13
xdr_reference	-	6.3.5.7
-	struct	6.6.6

6.7 Advanced Topics

This section describes techniques for passing data structures that are not covered in the preceding sections. Such structures include linked lists (of arbitrary lengths). Unlike the simpler examples covered in the previous sections, the following examples are written using both the XDR C library routines and the XDR data description language. Section 6.6 describes the XDR data definition language used below.

6.7.1 Linked Lists

The last example in section 6.2 presented a C data structure and its associated XDR routines for a person's gross assets and liabilities. The example is duplicated below:

```
struct gnumbers {
        long g_assets;
        long g_liabilities;
};

bool_t
xdr_gnumbers(xdrs, gp)
        XDR *xdrs;
        struct gnumbers *gp;
{
        if (xdr_long(xdrs, &(gp->g_assets)))
                return (xdr_long(xdrs, &(gp->g_liabilities)));
        return (FALSE);
}
```

Now assume that it is wished to implement a linked list of such information. A data structure could be constructed as follows:

```
typedef struct gnnode {
        struct gnumbers gn_numbers;
        struct gnnode *nxt;
};

typedef struct gnnode *gnumbers_list;
```

The head of the linked list can be thought of as the data object; that is, the head is not merely a convenient shorthand for a structure. Similarly, the `nxt` field is used to indicate whether or not the object has terminated. Unfortunately, if the object continues, the `nxt` field is also the address of where it continues. The link addresses carry no useful information when the object is serialized.

The XDR data description of this linked list is described by the recursive type declaration of `gnumbers_list`:

```
struct gnumbers {
        unsigned g_assets;
        unsigned g_liabilities;
};

typedef union switch (boolean) {
        case TRUE: struct {
                struct gnumbers current_element;
                gnumbers_list rest_of_list;
        };
        case FALSE: struct {};
} gnumbers_list;
```

In this description, the boolean indicates whether there is more data following it. If the boolean is `FALSE`, then it is the last data field of the structure. If it is `TRUE`, then it is followed by a `gnumbers`

structure and (recursively) by a `gnumbers_list` (the rest of the object). Note that the C declaration has no boolean explicitly declared in it (though the `nxt` field implicitly carries the information), while the XDR data description has no pointer explicitly declared in it.

Hints for writing a set of XDR routines to successfully (de)serialize a linked list of entries can be taken from the XDR description of the pointer-less data. The set consists of the following mutually recursive routines: `xdr_gnumbers_list`, `xdr_wrap_list`, and `xdr_gnnode`.

```
bool_t
xdr_gnnode(xdrs, gp)
        XDR *xdrs;
        struct gnnode *gp;
{
        return (xdr_gnumbers(xdrs, &(gp->gn_numbers)) &&
                xdr_gnumbers_list(xdrs, &(gp->nxt)));
}

bool_t
xdr_wrap_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        return (xdr_reference(xdrs, glp, sizeof(struct gnnode),
            xdr_gnnode));
}

struct xdr_discrim choices[2] = {
        /* called if another node needs (de)serializing */
        { TRUE, xdr_wrap_list },
         /* called when there are no more nodes to be */
         /* (de)serialized */
        { FALSE, xdr_void }
}

bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        more_data = (*glp != (gnumbers_list)NULL);
        return (xdr_union(xdrs, &more_data, glp, choices, NULL));
}
```

The entry routine is `xdr_gnumbers_list()`; its job is to translate between the boolean value `more_data` and the list pointer values. If there is no more data, the `xdr_union()` primitive calls `xdr_void()`, and the recursion is terminated. Otherwise, `xdr_union()` calls `xdr_wrap_list()`, whose job is to dereference the list pointers. The `xdr_gnnode()` routine actually (de)serializes data of the current node of the linked list and

recursively calls `xdr_gnumbers_list()` to handle the remainder of the list.

Readers should convince themselves that these routines function correctly in all three directions (`XDR_ENCODE`, `XDR_DECODE`, and `XDR_FREE`) for linked lists of any length (including zero). Note that the boolean `more_data` is always initialized, but in the `XDR_DECODE` case it is overwritten by an externally generated value. Also note that the value of the `bool_t` is lost in the stack. The essence of the value is reflected in the list's pointers.

The unfortunate side effect of (de)serializing a list with these routines is that the C stack grows linearly with respect to the number of nodes in the list. This is due to the recursion. The routines are also hard to code (and understand) due to the number and nature of the primitives involved (such as `xdr_reference`, `xdr_union`, and `xdr_void`).

The following routine collapses the recursive routines. It also has other optimizations that are discussed below.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;

        while (TRUE) {
             more_data = (*glp != (gnumbers_list)NULL);
             if (! xdr_bool(xdrs, &more_data))
                     return (FALSE);
             if (! more_data)
                     return (TRUE);  /* we are done */
             if (! xdr_reference(xdrs, glp, sizeof(struct gnnode),
                 xdr_gnumbers))
                     return (FALSE);
             glp = &((*glp)->nxt);
        }
}
```

The claim is that this one routine is easier to code and understand than the three recursive routines above. The parameter `glp` is treated as the address of the pointer to the head of the remainder of the list to be (de)serialized. Thus, `glp` is set to the address of the current node's `nxt` field at the end of the while loop. The discriminated union is implemented in-line; the variable `more_data` has the same use in this routine as in the routines above. Its value is recomputed and re-(de)serialized each iteration of the loop. Since `*glp` is a pointer to a node, the pointer is dereferenced using `xdr_reference()`. Note that the third parameter is truly the

size of a node (data values plus `nxt` pointer), while `xdr_gnumbers()` only (de)serializes the data values. This optimization works only because the `nxt` data comes after all legitimate external data.

There is a bug in this routine in the `XDR_FREE` case, in that `xdr_reference()` will free the node `*glp`. Upon return the assignment `glp = &((*glp)->nxt)` cannot be guaranteed to work since `*glp` is no longer a legitimate node. The following code works in all cases. The hard part is to avoid dereferencing a pointer which has not been initialized or which has been freed.

```
bool_t
xdr_gnumbers_list(xdrs, glp)
        XDR *xdrs;
        gnumbers_list *glp;
{
        bool_t more_data;
        bool_t freeing;
        gnumbers_list *next;  /* the next value of glp */

        freeing = (xdrs->x_op == XDR_FREE);
        while (TRUE) {
             more_data = (*glp != (gnumbers_list)NULL);
             if (! xdr_bool(xdrs, &more_data))
                     return (FALSE);
             if (! more_data)
                     return (TRUE);  /* we are done */
             if (freeing)
                     next = &((*glp)->nxt);
             if (! xdr_reference(xdrs, glp, sizeof(struct gnnode),
                 xdr_gnumbers))
                     return (FALSE);
             glp = (freeing) ? next : &((*glp)->nxt);
        }
}
```

Note that this is the first example in this document that actually inspects the direction of the operation (`xdrs->x_op`). The claim is that the correct iterative implementation is still easier to understand or code than the recursive implementation. It is certainly more efficient with respect to C stack requirements.

6.8 The Record Marking Standard

A record is composed of one or more record fragments. A record fragment is a 4-byte header followed by 0 *to* $2^{31}-1$ bytes of fragment data. The bytes encode an unsigned binary number; as with XDR integers, the byte order is from highest to lowest. The number encodes two values – a boolean that indicates whether the fragment is the last fragment of the record (bit value 1 implies the fragment is the last fragment) and a 31-bit unsigned binary value, which is the length in bytes of the fragment's data. The boolean value is the

high-order bit of the header; the length is made up of the 31 low-order bits.

Note that this record specification is *not* in XDR standard form and cannot be implemented using XDR primitives.

6.9 Synopsis of XDR Routines

xdr_array()

```
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
        XDR *xdrs;
        char **arrp;
        uint *sizep, maxsize, elsize;
        xdrproc_t elproc;
```

A filter primitive that translates between arrays and their corresponding external representations. The parameter `arrp` is the address of the pointer to the array, while `sizep` is the address of the element count of the array; this element count cannot exceed `maxsize`. The parameter `elsize` is the `sizeof()` each of the array's elements, and `elproc` is an XDR filter that translates between the array elements' C form and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_bool()

```
xdr_bool(xdrs, bp)
        XDR *xdrs;
        bool_t *bp;
```

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either 1 or 0. This routine returns 1 if it succeeds, 0 otherwise.

xdr_bytes()

```
xdr_bytes(xdrs, sp, sizep, maxsize)
        XDR *xdrs;
        char **sp;
        uint *sizep, maxsize;
```

A filter primitive that translates between counted byte strings and their external representations. The parameter `sp` is the address of the string pointer. The length of the string is located at address `sizep`; strings cannot be longer than `maxsize`. This routine returns 1 if it succeeds, 0 otherwise.

xdr_destroy()

```
void
xdr_destroy(xdrs)
        XDR *xdrs;
```

A macro that invokes the destroy routine associated with the XDR stream, `xdrs`. Destruction usually involves freeing private data structures associated with the stream. Using `xdrs` after invoking `xdr_destroy()` is undefined.

xdr_double()

```
xdr_double(xdrs, dp)
        XDR *xdrs;
        double *dp;
```

A filter primitive that translates between C `double` precision numbers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_enum()

```
xdr_enum(xdrs, ep)
        XDR *xdrs;
        enum_t *ep;
```

A filter primitive that translates between C `enums` (actually integers) and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_float()

```
xdr_float(xdrs, fp)
        XDR *xdrs;
        float *fp;
```

A filter primitive that translates between C single precision numbers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_getpos()

```
uint
xdr_getpos(xdrs)
        XDR *xdrs;
```

A macro that invokes the get-position routine associated with the XDR stream, `xdrs`. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

xdr_inline( )

```
long *
xdr_inline(xdrs, len)
        XDR *xdrs;
        int len;
```

A macro that invokes the in-line routine associated with the XDR stream, `xdrs`. The routine returns a pointer to a contiguous piece of the stream's buffer; `len` is the byte length of the desired buffer. Note that the pointer is cast to `long *`.

WARNING: `xdr_inline( )` may return 0 (`NULL`) if it cannot allocate a contiguous piece of a buffer. Therefore, the behavior may vary among stream instances; it exists for the sake of efficiency.

xdr_int( )

```
xdr_int(xdrs, ip)
        XDR *xdrs;
        int *ip;
```

A filter primitive that translates between C integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_long( )

```
xdr_long(xdrs, lp)
        XDR *xdrs;
        long *lp;
```

A filter primitive that translates between C `long` integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_opaque( )

```
xdr_opaque(xdrs, cp, cnt)
        XDR *xdrs;
        char *cp;
        uint cnt;
```

A filter primitive that translates between fixed size opaque data and its external representation. The parameter `cp` is the address of the opaque object, and `cnt` is its size in bytes. This routine returns 1 if it succeeds, 0 otherwise.

xdr_reference( )

```
xdr_reference(xdrs, pp, size, proc)
        XDR *xdrs;
        char **pp;
        uint size;
        xdrproc_t proc;
```

A primitive that provides pointer chasing within structures. The parameter `pp` is the address of the pointer, `size` is the `sizeof( )` the structure that `*pp` points to, and `proc` is an XDR procedure that filters the structure between its C form and its external representation. This routine returns 1 if it succeeds, 0 otherwise.

xdr_setpos( )

```
xdr_setpos(xdrs, pos)
        XDR *xdrs;
        uint pos;
```

A macro that invokes the set position routine associated with the XDR stream, `xdrs`. The parameter `pos` is a position value obtained from `xdr_getpos( )`. This routine returns 1 if the XDR stream could be repositioned, 0 otherwise.

WARNING: Since it is difficult to reposition some types of XDR streams, this routine may fail with one type of stream and succeed with another.

xdr_short( )

```
xdr_short(xdrs, sp)
        XDR *xdrs;
        short *sp;
```

A filter primitive that translates between C `short` integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_string( )

```
xdr_string(xdrs, sp, maxsize)
        XDR *xdrs;
        char **sp;
        uint maxsize;
```

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than `maxsize`. Note that `sp` is the address of the string's pointer. This routine returns 1 if it succeeds, 0 otherwise.

xdr_u_int( )

```
xdr_u_int(xdrs, up)
        XDR *xdrs;
        unsigned *up;
```

A filter primitive that translates between C `unsigned` integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_u_long( )

```
xdr_u_long(xdrs, ulp)
        XDR *xdrs;
        unsigned long *ulp;
```

A filter primitive that translates between C `unsigned long` integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_u_short( )

```
xdr_u_short(xdrs, usp)
        XDR *xdrs;
        unsigned short *usp;
```

A filter primitive that translates between C `unsigned short` integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_union( )

```
xdr_union(xdrs, dscmp, unp, choices, dfault)
        XDR *xdrs;
        int *dscmp;
        char *unp;
        struct xdr_discrim *choices;
        xdrproc_t dfault;
```

A filter primitive that translates between a discriminated C `union` and its corresponding external representation. The parameter `dscmp` is the address of the union's discriminant, while `unp` is the address of the union. This routine returns 1 if it succeeds, 0 otherwise.

xdr_void( )

```
xdr_void()
```

This routine always returns 1. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

xdr_wrapstring( )

```
xdr_wrapstring(xdrs, sp)
        XDR *xdrs;
        char **sp;
```

A primitive that calls `xdr_string(xdrs, sp, MAXUNSIGNED)`, where `MAXUNSIGNED` is the maximum value of an unsigned integer. This is useful because the RPC package passes only two parameters to XDR routines, whereas `xdr_string( )`, one of the most frequently used primitives, requires three parameters. This routine returns 1 if it succeeds, 0 otherwise.

xdrmem_create( )

```
void
xdrmem_create(xdrs, addr, size, op)
        XDR *xdrs;
        char *addr;
        uint size;
        enum xdr_op op;
```

This routine initializes the XDR stream object pointed to by `xdrs`. The stream's data is written to, or read from, a chunk of memory at location `addr`, whose length is no more than `size` bytes long. The `op` determines the direction of the XDR stream (either `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`).

xdrrec_create( )

```
void
xdrrec_create(xdrs, sendsize, recvsize, handle, readit,
writeit)
        XDR *xdrs;
        uint sendsize, recvsize;
        char *handle;
        int (*readit)(), (*writeit)();
```

This routine initializes the XDR stream object pointed to by `xdrs`. The stream's data is written to a buffer of size `sendsize`; a value of 0 indicates that the system should use a suitable default. The stream's data is read from a buffer of size `recvsize`; it too can be set to a suitable default by passing a 0 value. When a stream's output buffer is full, `writeit( )` is called. Similarly, when a stream's input buffer is empty, `readit( )` is called. The behavior of these two routines is similar to the INTERACTIVE UNIX Operating System system calls `read` and `write`, except that `handle`

is passed to the former routines as the first parameter. Note that the XDR stream's `op` field must be set by the caller.

WARNING: This XDR stream implements an intermediate record stream. There are, therefore, additional bytes in the stream to provide record boundary information.

xdrrec_endofrecord( )

```
xdrrec_endofrecord(xdrs, sendnow)
        XDR *xdrs;
        int sendnow;
```

This routine can be invoked only on streams created by `xdrrec_create( )`. The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if `sendnow` is nonzero. This routine returns 1 if it succeeds, 0 otherwise.

xdrrec_eof( )

```
xdrrec_eof(xdrs)
        XDR *xdrs;
        int empty;
```

This routine can be invoked only on streams created by `xdrrec_create( )`. After consuming the rest of the current record in the stream, this routine returns 1 if the stream has no more input, 0 otherwise.

xdrrec_skiprecord( )

```
xdrrec_skiprecord(xdrs)
        XDR *xdrs;
```

This routine can be invoked only on streams created by `xdrrec_create( )`. It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns 1 if it succeeds, 0 otherwise.

xdrstdio_create( )

```
void
xdrstdio_create(xdrs, file, op)
        XDR *xdrs;
        FILE *file;
        enum xdr_op op;
```

This routine initializes the XDR stream object pointed to by `xdrs`. The XDR stream data is written to, or read from, the standard I/O stream `file`. The parameter `op` determines

the direction of the XDR stream (either `XDR_ENCODE`, `XDR_DECODE`, or `XDR_FREE`).

WARNING: The destroy routine associated with such XDR streams calls `fflush()` on the `file` stream, but never `fclose()`.

7. NFS PROTOCOL SPECIFICATION

7.1 Introduction

The Network File System (NFS) protocol provides transparent remote access to shared file systems over local area networks. The NFS protocol is designed to be machine, operating system, network architecture, and transport protocol independent. This independence is achieved through the use of Remote Procedure Call (RPC) primitives built on top of an eXternal Data Representation (XDR).

The supporting mount protocol allows the server to hand out remote access privileges to a restricted set of clients. Thus, it allows clients to attach a remote directory tree at any point on some local file system.

7.1.1 Remote Procedure Call

The remote procedure call specification, described in section 4 provides a clean, procedure-oriented interface to remote services. Each server supplies a program that is a set of procedures. The combination of host address, program number, and procedure number specifies one remote service procedure.

RPC is a high-level protocol built on top of low-level transport protocols. Since it does not depend on services provided by specific protocols, it can be used easily with any underlying transport protocol. The only transport protocol currently supported is UDP/IP.

The RPC protocol includes a slot for authentication parameters on every call. The contents of the authentication parameters are determined by the "flavor" (type) of authentication used by the server and client. A server may support several different flavors of authentication at once: `AUTH_NONE` passes no authentication information (this is called null authentication); `AUTH_UNIX` passes the INTERACTIVE UNIX System `uid`, `gid`, and `groups` with each call.

Servers may change over time, and the protocol which they use may change, too, so RPC provides a version number with each RPC request. Thus, one server can service requests for several different versions of the protocol at the same time.

7.1.2 eXternal Data Representation

The eXternal Data Representation specification, described in section 6 provides a common way of representing a set of data types over a

network. This takes care of problems such as different byte ordering on different communicating machines. It also defines the size of each data type so that machines with different structure alignment algorithms can share a common format over the network.

In this section the XDR data definition language is used to specify the parameters and results of each RPC service procedure that a NFS server provides. The XDR data definition language is similar to C, although a few new constructs have been added. The notation

```
string name[SIZE];
string data<DSIZE>;
```

defines `name`, which is a fixed size block of `SIZE` bytes, and `data`, which is a variable size block of up to `DSIZE` bytes. This same notation is used to indicate fixed-length arrays and arrays with a variable number of elements up to some maximum.

The discriminated union definition

```
union switch (enum status) {
        NFS_OK:
                struct {
                        filename file1;
                        filename file2;
                        integer count;
                }
        NFS_ERROR:
                struct {
                        errstat error;
                        integer errno;
                }
        default:
                struct {}
}
```

means the first thing over the network is an enumeration type called `status`; if its value is `NFS_OK`, the next thing on the network will be the structure containing `file1`, `file2`, and `count`. If the value of `status` is neither `NFS_OK` nor `NFS_ERROR`, then there is no more data to look at.

7.1.3 Stateless Servers

The NFS protocol is stateless. That is, a server does not need to maintain state about any of its clients in order to function correctly. Stateless servers have a distinct advantage over stateful servers in the event of a crash. With stateless servers, a client need only retry a request until the server responds; it does not even need to know that the server has crashed. The client of a stateful server, on the other hand, needs to detect a server crash and rebuild the server's state when it comes back up.

This issue may not seem important, but it affects the protocol in several ways. It is worth extra complexity in the protocol to be able to write very simple servers with no crash recovery provisions.

7.2 NFS Protocol Definition

The NFS protocol is designed to be operating system independent, but it was designed in a UNIX System environment. As such, it has some features that are very much like the INTERACTIVE UNIX Operating System. When in doubt about how something should work, referring to the way it is done on the INTERACTIVE UNIX Operating System should provide assistance.

The protocol definition is given as a set of procedures with arguments and results defined using XDR. A brief description of the function of each procedure should provide enough information to allow implementation on most machines. A different section is provided for each supported version of the protocol. Most of the procedures, and their parameters and results, are self-explanatory. However, a few do not fit into the normal UNIX System mold.

The `LOOKUP` procedure looks up one component of a path name at a time. It is not immediately obvious why it does not take the whole path name, look down the directories, and return a file handle when it is done. There are two good reasons not to do this. First, path names need separators between the directory components, and different operating systems use different separators. A Network Standard Pathname Representation could be defined, but in this case every path name would have to be parsed and converted at each end. Second, if path names were passed, the server would have to keep track of the mounted file systems for all of its clients, so that it could break the path name at the right point and pass the remainder on to the correct server.

Another procedure that might seem strange is the `READDIR` procedure. `READDIR` provides a network standard format for representing directories. The argument above could have been used to justify a `READDIR` procedure that returns only one directory entry per call. The problem is efficiency – directories can contain many entries, and a remote call to return each would be too slow.

7.2.1 Server/Client Relationship

The NFS protocol is designed to allow servers to be as simple and general as possible. Sometimes the simplicity of the server can be a

problem if the client wants to implement complicated file system semantics.

For example, the INTERACTIVE UNIX System allows removal of open files. A process can open a file and, while it is open, remove it from the directory. The file can be read and written as long as the process keeps it open, even though the file has no name in the file system. It is impossible for a stateless server to implement these semantics. The client can perform some functions, such as renaming the file on remove and only removing it on close. It is felt that the server provides enough functionality to implement most file system semantics on the client.

Every NFS client can also be a server, and remote and local mounted file systems can be freely intermixed. This leads to some interesting problems when a client travels down the directory tree of a remote file system and reaches the mount point on the server for another remote file system. Allowing the server to follow the second remote mount means it must do loop detection, server lookup, and user revalidation.

Instead, it was decided not to let clients cross a server's mount point. When a client does a `LOOKUP` on a directory on which the server has mounted a file system, the client sees the underlying directory instead of the mounted directory. A client can do remote mounts that match the server's mount points to maintain the server's view.

7.2.2 Permission Issues

The NFS protocol, strictly speaking, does not define the permission checking used by servers. However, it is expected that a server will do normal INTERACTIVE UNIX System permission checking using `AUTH_UNIX` style authentication as the basis of its protection mechanism. The server gets the client's effective `uid` and effective `gid` and groups on each call, and uses them to check permission. Various problems with this method can be resolved in interesting ways.

Using `uid` and `gid` implies that the client and server share the same `uid` list. Every server and client pair must have the same mapping from user to `uid` and from group to `gid`. Since every client can also be a server, this tends to imply that the whole network shares the same `uid/gid` space. This is acceptable for the

short term, but a more workable network authentication method will be necessary before long.

Another problem arises due to the semantics of `open`. The UNIX Operating System does its permission checking at open time and then assumes that the file is open and has been checked on later read and write requests. With stateless servers this breaks down, because the server has no idea that the file is open and it must do permission checking on each read and write call. On a local file system, a user can open a file and then change the permissions so that no one is allowed to touch it, but will still be able to write to the file because it is open. On a remote file system, by contrast, the write would fail. To get around this problem, the server's permission checking algorithm should allow the owner of a file to access it no matter what the permissions are set to.

A similar problem has to do with paging in from a file over the network. The INTERACTIVE UNIX System kernel checks for execute permission before opening a file for demand paging, then reads blocks from the open file. The file may not have read permission, but after it is opened this does not matter. An NFS server cannot tell the difference between a normal file read and a demand page-in read. To make this work, the server allows reading of files if the `uid` given in the call has execute or read permission on the file.

In the INTERACTIVE UNIX Operating System, the user ID zero has access to all files no matter what permission and ownership they have. This superuser permission is not allowed on the server since anyone who can become superuser on his own machine could gain access to all remote files. Instead, the server maps `uid` 0 to −2 before doing its access checking. This works as long as NFS is not used to supply `root` file systems, where superuser access cannot be avoided. Eventually servers will have to allow some kind of limited superuser access.

7.2.3 RPC Information

Authentication

The NFS service uses `AUTH_UNIX` style authentication except in the `NULL` procedure where `AUTH_NONE` is also allowed.

Protocols

NFS currently is supported on UDP/IP only.

Constants
These are the RPC constants needed to call the NFS service. They are given in decimal.

PROGRAM	100003
VERSION	2

Port Number
The NFS protocol currently uses the UDP port number 2049. This restriction will be waived in a future protocol revision.

7.2.4 Sizes

These are the sizes, given in decimal bytes, of various XDR structures used in the protocol.

MAXDATA 8192
The maximum number of bytes of data in a `READ` or `WRITE` request.

MAXPATHLEN 1024
The maximum number of bytes in a path name argument.

MAXNAMLEN 255
The maximum number of bytes in a file name argument.

COOKIESIZE 4
The size in bytes of the opaque "cookie" passed by `READDIR`.

FHSIZE 32
The size in bytes of the opaque file handle.

7.2.5 Basic Data Types

The following XDR definitions are basic structures and types used in other structures later on.

7.2.5.1 `stat`.

```
typedef enum {
        NFS_OK=0,
        NFSERR_PERM=1,
        NFSERR_NOENT=2,
        NFSERR_IO=5,
        NFSERR_NXIO=6,
        NFSERR_ACCES=13,
        NFSERR_EXIST=17,
        NFSERR_NODEV=19,
        NFSERR_NOTDIR=20,
        NFSERR_ISDIR=21,
        NFSERR_FBIG=27,
        NFSERR_NOSPC=28,
        NFSERR_ROFS=30,
        NFSERR_NAMETOOLONG=63,
        NFSERR_NOTEMPTY=66,
        NFSERR_DQUOT=69,
        NFSERR_STALE=70,
        NFSERR_WFLUSH=99
} stat;
```

The `stat` type is returned with every procedure's results. A value of `NFS_OK` indicates that the call completed successfully and the results are valid. The other values indicate that some kind of error occurred on the server side during the servicing of the procedure. The error values are derived from UNIX System error numbers.

NFSERR_PERM	Not owner. The caller does not have correct ownership to perform the requested operation.
NFSERR_NOENT	No such file or directory. The file or directory specified does not exist.
NFSERR_IO	I/O error. A hard error, for example a disk error, occurred when the operation was in progress.
NFSERR_NXIO	No such device or address.
NFSERR_ACCES	Permission denied. The caller does not have the correct permission to perform the requested operation.
NFSERR_EXIST	File exists. The file specified already exists.
NFSERR_NODEV	No such device.
NFSERR_NOTDIR	Not a directory. The caller specified a non-directory in a directory operation.

NFSERR_ISDIR	Is a directory. The caller specified a directory in a non-directory operation.
NFSERR_FBIG	File too large. The operation caused a file to grow beyond the server's limit.
NFSERR_NOSPC	No space left on device. The operation caused the server's file system to reach its limit.
NFSERR_ROFS	Read-only file system. Write attempted on a read-only file system.
NFSERR_NAMETOOLONG	File name too long. The file name in an operation was too long.
NFSERR_NOTEMPTY	Directory not empty. Attempted to remove a directory that was not empty.
NFSERR_DQUOT	Disk quota exceeded. The client's disk quota on the server has been exceeded.
NFSERR_STALE	The `fhandle` given in the arguments was invalid. That is, the file referred to by that file handle no longer exists or access to it has been revoked.
NFSERR_WFLUSH	The server's write cache used in the `WRITECACHE` call got flushed to disk.

7.2.5.2 `ftype`.

```
typedef enum {
        NFNON=0,
        NFREG=1,
        NFDIR=2,
        NFBLK=3,
        NFCHR=4,
        NFLNK=5
} ftype;
```

The enumeration `ftype` gives the type of a file. The type `NFNON` indicates a non-file, `NFREG` is a regular file, `NFDIR` is a directory, `NFBLK` is a block-special device, `NFCHR` is a character-special device, and `NFLNK` is a symbolic link.

7.2.5.3 `fhandle`.

```
typedef opaque fhandle[FHSIZE];
```

The `fhandle` is the file handle that the server passes to the client. All file operations are done using file handles to refer to a file or

directory. The file handle can contain whatever information the server needs to distinguish an individual file.

7.2.5.4 `timeval`.

```
typedef struct {
        unsigned seconds;
        unsigned useconds;
} timeval;
```

The `timeval` structure is the number of seconds and microseconds since midnight on January 1, 1970 Greenwich Mean Time. It is used to pass time and date information.

7.2.5.5 `fattr`.

```
typedef struct {
        ftype type;
        unsigned mode;
        unsigned nlink;
        unsigned uid;
        unsigned gid;
        unsigned size;
        unsigned blocksize;
        unsigned rdev;
        unsigned blocks;
        unsigned fsid;
        unsigned fileid;
        timeval atime;
        timeval mtime;
        timeval ctime;
} fattr;
```

The `fattr` structure contains the attributes of a file; `type` is the type of the file; `nlink` is the number of hard links to the file, that is, the number of different names for the same file; `uid` is the user identification number of the owner of the file; `gid` is the group identification number of the group of the file; `size` is the size in bytes of the file; `blocksize` is the size in bytes of a block of the file; `rdev` is the device number of the file if it is type `NFCHR` or `NFBLK`; `blocks` is the number of blocks that the file takes up on disk; `fsid` is the file system identifier for the file system that contains the file; `fileid` is a number that uniquely identifies the file within its file system; `atime` is the time when the file was last accessed for either read or write; `mtime` is the time when the file data was last modified (written); and `ctime` is the time when the status of the file was last changed. Writing to the file also changes `ctime` if the size of the file changes.

The `mode` field is the access mode encoded as a set of bits. The bits are the same as the mode bits returned by the *stat*(2) system call in the UNIX Operating System. Notice that the file type is

specified both in the mode bits and in the file type. This will be fixed in future versions. The descriptions given below specify the bit positions using octal numbers.

0040000	This is a directory. The `type` field should be `NFDIR`.
0020000	This is a character special file. The `type` field should be `NFCHR`.
0060000	This is a block special file. The `type` field should be `NFBLK`.
0100000	This is a regular file. The `type` field should be `NFREG`.
0120000	This is a symbolic link file. The `type` field should be `NFLNK`.
0140000	This is a named socket. The `type` field should be `NFNON`.
0004000	Set user ID on execution.
0002000	Set group ID on execution.
0001000	Save swapped text even after use.
0000400	Read permission for owner.
0000200	Write permission for owner.
0000100	Execute and search permission for owner.
0000040	Read permission for group.
0000020	Write permission for group.
0000010	Execute and search permission for group.
0000004	Read permission for others.
0000002	Write permission for others.
0000001	Execute and search permission for others.

7.2.5.6 `sattr`.

```
typedef struct {
        unsigned mode;
        unsigned uid;
        unsigned gid;
        unsigned size;
        timeval atime;
        timeval mtime;
} sattr;
```

The `sattr` structure contains the file attributes that can be set from the client. The fields are the same as for `fattr` above. A `size` of zero means the file should be truncated. A value of -1 indicates a field that should be ignored.

7.2.5.7 filename.

```
typedef string filename<MAXNAMLEN>;
```

The type `filename` is used for passing file names or path name components.

7.2.5.8 path.

```
typedef string path<MAXPATHLEN>;
```

The type `path` is a path name. The server considers it as a string with no internal structure, but to the client it is the name of a node in a file system tree.

7.2.5.9 attrstat.

```
typedef union switch (stat status) {
        NFS_OK:
                fattr attributes;
        default:
                struct {}
} attrstat;
```

The `attrstat` structure is a common procedure result. It contains a `status` and, if the call succeeded, it also contains the attributes of the file on which the operation was done.

7.2.5.10 diropargs.

```
typedef struct {
        fhandle dir;
        filename name;
} diropargs;
```

The `diropargs` structure is used in directory operations. The file handle `dir` is the directory in which to find the file `name`. A directory operation is one in which the directory is affected.

7.2.5.11 diropres.

```
typedef union switch (stat status) {
        NFS_OK:
                struct {
                        fhandle file;
                        fattr attributes;
                }
        default:
                struct {}
} diropres;
```

The results of a directory operation are returned in a `diropres` structure. If the call succeeded, a new file handle `file` and the `attributes` associated with that file are returned along with the `status`.

7.2.6 Server Procedures

The following sections define the RPC procedures supplied by an NFS server. The RPC procedure number and version are given in the header, along with the name of the procedure. The synopsis of procedures has this format:

```
<proc #>. <proc name> ( <arguments> ) returns ( <results> )
          <argument declarations>
          <results declarations>
```

In the first line, `proc name` is the name of the procedure, `arguments` is a list of the names of the arguments, and `results` is a list of the names of the results. The second line gives the XDR `argument declarations` and the third line gives the XDR `results declarations`. Afterwards, there is a description of what the procedure is expected to do and how its arguments and results are used. If there are bugs or problems with the procedure, they are listed at the end.

All of the procedures in the NFS protocol are assumed to be synchronous. When a procedure returns to the client, the client can assume that the operation has completed and any data associated with the request is now on stable storage. For example, a client `WRITE` request may cause the server to update data blocks, file system information blocks (such as indirect blocks in the INTERACTIVE UNIX System), and file attribute information (size and modify times). When the `WRITE` returns to the client, it can assume that the write is safe, even in case of a server crash, and it can discard the data written. This is a very important part of the statelessness of the server. If the server waited to flush data from remote requests, the client would have to save those requests so that it could resend them in case of a server crash.

7.2.6.1 Do Nothing (Procedure 0, Version 2).

```
0. NFSPROC_NULL () returns ()
```

This procedure does no work. It is made available in all RPC services to allow server response testing and timing.

7.2.6.2 Get File Attributes (Procedure 1, Version 2).

```
1. NFSPROC_GETATTR (file) returns (reply)
        fhandle file;
        attrstat reply;
```

If `reply.status` is `NFS_OK`, then `reply.attributes` contains the attributes for the file given by `file`.

Bugs: The `rdev` field in the attributes structure is an INTERACTIVE UNIX System device specifier. It should be removed or generalized.

7.2.6.3 Set File Attributes (Procedure 2, Version 2).

```
2. NFSPROC_SETATTR (file, attributes) returns (reply)
        fhandle file;
        sattr attributes;
        attrstat reply;
```

The `attributes` argument contains fields that are either -1 or are the new value for the attributes of `file`. If `reply.status` is `NFS_OK`, then `reply.attributes` has the attributes of the file after the `setattr` operation has completed.

Bugs: The use of -1 to indicate an unused field in `attributes` is wrong.

7.2.6.4 Get File System Root (Procedure 3, Version 2).

```
3. NFSPROC_ROOT () returns ()
```

Obsolete. This procedure is no longer used because finding the `root` file handle of a file system requires moving path names between client and server. To do this correctly, it would be necessary to define a network standard representation of path names. Instead, the function of looking up the `root` file handle is done by the `MNTPROC_MNT` procedure (see section 7.3 for details).

7.2.6.5 Look Up File Name (Procedure 4, Version 2).

```
4. NFSPROC_LOOKUP (which) returns (reply)
        diropargs which;
        dirоpres reply;
```

If `reply.status` is `NFS_OK`, then `reply.file` and `reply.attributes` are the file handle and attributes for the file `which.name` in the directory given by `which.dir`.

Bugs: There is some question as to what is the correct reply to a `LOOKUP` request when `which.name` is a mount point on the server for a remote mounted file system. Currently, the `fhandle` of the underlying directory is returned. This is not completely acceptable, as the clients see a different view of the file system to that seen by the server.

7.2.6.6 Read From Symbolic Link (Procedure 5, Version 2).

```
5. NFSPROC_READLINK (file) returns (reply)
        fhandle file;
        union switch (stat status) {
                NFS_OK:
                        path data;
                default:
                        struct {}
        } reply;
```

If `status` has the value `NFS_OK`, then `reply.data` is the data in the symbolic link given by `file`.

7.2.6.7 Read From File (Procedure 6, Version 2).

```
6. NFSPROC_READ (file, offset, count, totalcount) returns (reply)
        fhandle file;
        unsigned offset;
        unsigned count;
        unsigned totalcount;
        union switch (stat status) {
                NFS_OK:
                        fattr attributes;
                        string data<MAXDATA>;
                default:
                        struct {}
        } reply;
```

Returns up to `count` bytes of `data` from the file given by `file`, starting at `offset` bytes from the beginning of the file. The first byte of the file is at offset zero. The file attributes after the read takes place are returned in `attributes`.

Bugs: The argument `totalcount` is unused and should be removed.

7.2.6.8 Write to Cache (Procedure 7, Version 2).

```
7. NFSPROC_WRITECACHE () returns ()
```

Obsolete.

7.2.6.9 Write to File (Procedure 8, Version 2).

```
8. NFSPROC_WRITE (file, beginoffset, offset, totalcount, data)
   returns (reply)
        fhandle file;
        unsigned beginoffset;
        unsigned offset;
        unsigned totalcount;
        string data<MAXDATA>;
        attrstat reply;
```

Writes `data` beginning `offset` bytes from the beginning of `file`. The first byte of the file is at offset zero. If `reply.status` is `NFS_OK`, then `reply.attributes` contains the attributes of the file after the write has completed. The write operation is atomic. Data from this `WRITE` will not be mixed with data from another client's `WRITE`.

Bugs: The arguments `beginoffset` and `totalcount` are ignored and should be removed.

7.2.6.10 Create File (Procedure 9, Version 2).

```
9. NFSPROC_CREATE (where, attributes) returns (reply)
        diropargs where;
        sattr attributes;
        diropres reply;
```

The file `where.name` is created in the directory given by `where.dir`. The initial attributes of the new file are given by `attributes`. A `reply.status` of `NFS_OK` indicates that the file was created and that `reply.file` and `reply.attributes` are its file handle and attributes. Any other `reply.status` means that the operation failed and that no file was created.

Bugs: This routine should pass an exclusive `create` flag meaning "create the file only if it is not already there."

7.2.6.11 Remove File (Procedure 10, Version 2).

```
10. NFSPROC_REMOVE (which) returns (status)
        diropargs which;
        stat status;
```

The file `which.name` is removed from the directory given by `which.dir`. A `status` of `NFS_OK` means the directory entry was removed.

7.2.6.12 Rename File (Procedure 11, Version 2).

```
11. NFSPROC_RENAME (from, to) returns (status)
        diropargs from;
        diropargs to;
        stat status;
```

The existing file `from.name` in the directory given by `from.dir` is renamed to `to.name` in the directory given by `to.dir`. If `status` is `NFS_OK`, the file was renamed. The `RENAME` operation is atomic on the server; it cannot be interrupted in the middle.

7.2.6.13 Create Link to File (Procedure 12, Version 2).

```
12. NFSPROC_LINK (from, to) returns (status)
        fhandle from;
        diropargs to;
        stat status;
```

Creates the file `to.name` in the directory given by `to.dir`, which is a hard link to the existing file given by `from`. If the return value of `status` is `NFS_OK`, a link was created. Any other return value indicates an error and the link is not created.

A hard link should ensure that changes to either of the linked files are reflected in both files. When a hard link is made to a file, the attributes for the file should have a value for `nlink` which is one greater than the value before the link.

7.2.6.14 Create Symbolic Link (Procedure 13, Version 2).

```
13. NFSPROC_SYMLINK (from, to, attributes) returns (status)
        diropargs from;
        path to;
        sattr attributes;
        stat status;
```

Creates the file `from.name` with ftype `NFLNK` in the directory given by `from.dir`. The new file contains the path name `to` and has initial attributes given by `attributes`. If the return value of `status` is `NFS_OK`, a link was created. Any other return value indicates an error and the link is not created.

A symbolic link is a pointer to another file. The name given in `to` is not interpreted by the server, just stored in the newly created file. A `READLINK` operation returns the data to the client for interpretation.

7.2.6.15 Create Directory (Procedure 14, Version 2).

```
14. NFSPROC_MKDIR (where, attributes) returns (reply)
        diropargs where;
        sattr attributes;
        diropres reply;
```

The new directory `where.name` is created in the directory given by `where.dir`. The initial attributes of the new directory are given by `attributes`. A `reply.status` of `NFS_OK` indicates that the new directory was created and that `reply.file` and `reply.attributes` are its file handle and attributes. Any other `reply.status` means that the operation failed and that no directory was created.

7.2.6.16 Remove Directory (Procedure 15, Version 2).

```
15. NFSPROC_RMDIR (which) returns (status)
        diropargs which;
        stat status;
```

The existing empty directory `which.name` in the directory given by `which.dir` is removed. If `status` is `NFS_OK`, the directory was removed.

7.2.6.17 Read From Directory (Procedure 16, Version 2).

```
16. NFSPROC_READDIR (dir, cookie, count) returns (entries)
      fhandle dir;
      opaque cookie[COOKIESIZE];
      unsigned count;
      union switch (stat status) {
            NFS_OK:
                  typedef union switch (boolean valid) {
                        TRUE:
                              struct {
                                    unsigned fileid;
                                    filename name;
                                    opaque cookie[COOKIESIZE];
                                    entry nextentry;
                              }
                        FALSE:
                              struct {}
                  } entry;
                  boolean eof;
            default:
                  struct {}
      } entries;
```

Returns a variable number of directory entries, with a total size of up to `count` bytes, from the directory given by `dir`. Each `entry` contains a `fileid` which is a unique number to identify the file within a file system, the `name` of the file, and a `cookie` which is an opaque pointer to the next entry in the directory. The cookie is used in the next `READDIR` call to get more entries

starting at a given point in the directory. The special cookie zero (all bits zero) can be used to get the entries starting at the beginning of the directory. The `fileid` field should be the same number as the `fileid` in the attributes of the file. The `eof` flag has a value of `TRUE` if there are no more entries in the directory; `FALSE` otherwise. The `valid` flag is used to indicate whether there are more entries in this reply message. If the returned value of `status` is `NFS_OK`, then it is followed by a variable number of `entries`.

7.2.6.18 Get File System Attributes (Procedure 17, Version 2).

```
17. NFSPROC_STATFS (file) returns (reply)
        fhandle file;
        union switch (stat status) {
                NFS_OK:
                        struct {
                                unsigned tsize;
                                unsigned bsize;
                                unsigned blocks;
                                unsigned bfree;
                                unsigned bavail;
                        } fsattr;
                default:
                        struct {}
        } reply;
```

If `reply.status` is `NFS_OK`, then `reply.fsattr` gives the attributes for the file system that contains `file`. The attribute fields contain the following values:

`tsize` The optimum transfer size of the server in bytes. This is the number of bytes the server would like to have in the data part of `READ` and `WRITE` requests.

`bsize` The block size in bytes of the file system.

`blocks` The total number of `bsize` blocks on the file system.

`bfree` The number of free `bsize` blocks on the file system.

`bavail` The number of `bsize` blocks available to non-privileged users.

Bugs: This call does not work well if a file system has variable size blocks.

7.3 Mount Protocol Definition

The mount protocol is separate from, but related to, the NFS protocol. It provides all of the operating system specific services to get NFS off the ground – looking up path names, validating user identity, and checking access permissions. Clients use the mount protocol to get the first file handle, which allows them entry into a remote file system.

The mount protocol is kept separate from the NFS protocol to make it easy to plug in new access checking and validation methods without changing the NFS server protocol.

Notice that the protocol definition implies stateful servers because the server maintains a list of a client's mount requests. The mount list information is not critical for the correct functioning of either the client or the server. It is intended for advisory use only; for example, to warn possible clients when a server is going down.

7.3.1 Version 1

Version one of the mount protocol communicates with version two of the NFS protocol. The only connecting point is the `fhandle` structure, which is the same for both protocols.

7.3.1.1 RPC Information.

Authentication

The mount service uses `AUTH_UNIX` style authentication only.

Protocols

The mount service is currently supported on UDP/IP only.

Constants

These are the RPC constants needed to call the `MOUNT` service. They are given in decimal.

PROGRAM	100005
VERSION	1

Port Number
The server's port mapper, described in section 5, should be consulted to find which port number the mount service is registered on.

7.3.1.2 Sizes. These are the sizes given in decimal bytes of various XDR structures used in the protocol.

MNTPATHLEN 1024
The maximum number of bytes in a path name argument.

MNTNAMLEN 255
The maximum number of bytes in a name argument.

FHSIZE 32
The size in bytes of the opaque file handle.

7.3.1.3 Basic Data Types.

`fhandle`

```
typedef opaque fhandle[FHSIZE];
```

The `fhandle` is the file handle that the server passes to the client. All file operations are done using file handles to refer to a file or directory. The file handle can contain whatever information the server needs to distinguish an individual file.

This is the same as the `fhandle` XDR definition in version 2 of the NFS protocol; see the section on `fhandle` under "Basic Data Types" in that version.

`fhstatus`

```
typedef union switch (unsigned status) {
        0:
                fhandle directory;
        default:
                struct {}
}
```

If a `status` of zero is returned, the call completed successfully and a file handle for the `directory` follows. A nonzero status indicates some sort of error. In this case the status is a UNIX System error number.

`dirpath`

```
typedef string dirpath<MNTPATHLEN>;
```

The type `dirpath` is a normal UNIX System path name of a directory.

`name`

```
typedef string name<MNTNAMLEN>;
```

The type `name` is an arbitrary string used for various names.

7.3.1.4 Server Procedures. The following sections define the RPC procedures supplied by a mount server. The RPC procedure number and version are given in the header, along with the name of the procedure. The synopsis of procedures has the format:

```
<proc #>. <proc name> ( <arguments> ) returns ( <results> )
          <argument declarations>
          <results declarations>
```

In the first line, `proc name` is the name of the procedure, `arguments` is a list of the names of the arguments, and `results` is a list of the names of the results. The second line gives the XDR `argument declarations` and the third line gives the XDR `results declarations`. Afterwards there is a description of what the procedure is expected to do and how its arguments and results are used. If there are bugs or problems with the procedure, they are listed at the end.

7.3.1.5 Do Nothing (Procedure 0, Version 1).

```
0. MNTPROC_NULL () returns ()
```

This procedure does no work. It is made available in all RPC services to allow server response testing and timing.

7.3.1.6 Add Mount Entry (Procedure 1, Version 1).

```
1. MNTPROC_MNT (directory) returns (reply)
        dirpath dirname;
        fhstatus reply;
```

If `reply.status` is 0, `reply.directory` contains the file handle for the directory `dirname`. This file handle may be used in the NFS protocol. This procedure also adds a new entry to the mount list for this client mounting `dirname`.

7.3.1.7 Return Mount Entries (Procedure 2, Version 1).

```
2. MNTPROC_DUMP () returns (mountlist)
        union switch (boolean more_entries) {
                TRUE:
                        struct {
                                name hostname;
                                dirpath directory;
                                mountlist nextentry;
                        }
                FALSE:
                        struct {}
        } mountlist;
```

Returns the list of remote mounted file systems. The `mountlist` contains one entry for each `hostname` and `directory` pair.

7.3.1.8 Remove Mount Entry (Procedure 3, Version 1).

```
3. MNTPROC_UMNT (directory) returns ()
        dirpath directory;
```

Removes the mount list entry for `directory`.

7.3.1.9 Remove All Mount Entries (Procedure 4, Version 1).

```
4. MNTPROC_UMNTALL () returns ()
```

Removes all of the mount list entries for this client.

7.3.1.10 Return Export List (Procedure 5, Version 1).

```
5. MNTPROC_EXPORT () returns (exportlist)
      union switch (boolean more_entries) {
           TRUE:
              struct {
                 dirpath filesys;
                 typedef union switch (boolean more_groups) {
                      TRUE:
                           struct {
                                name grname;
                                groups nextgroup;
                           }
                      FALSE:
                              struct {}
                 } groups;
                 exportlist nextentry;
              }
           FALSE:
                struct {}
      } exportlist;
```

Returns in `exportlist` a variable number of export list entries. Each entry contains a file system name and a list of groups that are allowed to import it. The file system name is in `exportlist.filesys`, and the group name is in `exportlist.groups.grname`.

Bugs: The `exportlist` should contain more information about the status of the file system, such as a read-only flag.

8. AUTOMOUNTER GUIDE

8.1 Introduction

Sections 1 and 2 of this document explain how to export and mount file systems through NFS with the `mount` command. You can also NFS mount file systems using the `automount` program, which enables users to mount and unmount remote directories on an as-needed basis. Whenever a user on a client machine running the automounter invokes a command that accesses a remote file or directory, such as opening a file with an editor, the file system to which that file or directory belongs is mounted and remains mounted for as long as it is needed. No mounting is done at boot time, and the user no longer has to know the superuser password to mount a directory. It is all done automatically and transparently.

The automounter determines which mount points to monitor and which file systems to mount from a special set of files called maps (see section 8.2.1, "Preparing the Maps"). These maps can reside on the local machine or be managed via NIS. When `automount` is started, either from the command line or from `/etc/nfs` it forks a daemon to serve the mount points specified in the maps. It does this by establishing itself as the NFS server for the specified mount points. When one of these mount points is accessed or crossed, the automounter fields the NFS protocol request as would the real NFS server daemon, *nfsd*(1M). It then mounts the appropriate remote file system, as specified in the automounter maps. When a predetermined amount of time has elapsed without the file system being accessed, the automounter automatically unmounts it.

The automounter actually mounts all file systems under the directory `/tmp_mnt`. It then uses the NFS symbolic link[1] support to associate the actual mount point with the one in `/tmp_mnt`. The result is that the file systems, actually mounted under `/tmp_mnt`, appear to be mounted on the correct mount points.

To illustrate, assume that the automounter was configured to mount the remote file system `bigmo:/usr/man` on the local directory

1. A symbolic link is conceptually similar to a standard System V *link*(2), but can be established between path names residing on different file systems.

`/usr/man`. The first user to issue the command `cd /usr/man` would cause the automounter to perform the *equivalent* of the following operations:

```
# mkdir /tmp_mnt/usr/man
# mount -f NFS bigmo:/usr/man /tmp_mnt/usr/man
# ln -s[2] /tmp_mnt/usr/man /usr/man
```

If the `/tmp_mnt` or `/tmp_mnt/usr` directories did not exist at the time, the automounter would dynamically `mkdir` them as well. And although the automounter does not really create symbolic links, it does act as a symbolic link server, in this case, redirecting access requests destined for `/usr/man` to `/tmp_mnt/usr/man`. As a result, `bigmo:/usr/man` would appear to be directly NFS-mounted on the local `/usr/man` directory.

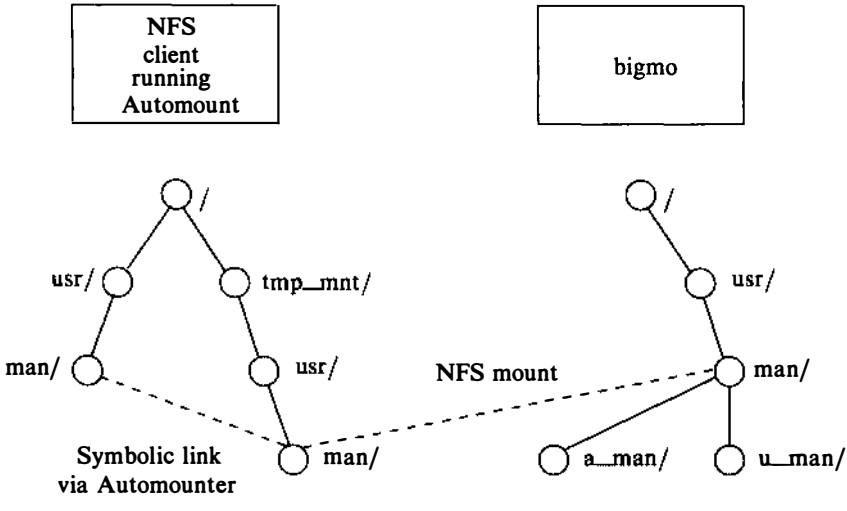


Figure 5. An Automounter Example

2. This option (not available on all versions of the operating system) is used to indicate a symbolic link operation, in this case, associating `/usr/man` and `/tmp_mnt/usr/man`.

This example shows how the automounter can be used to automatically mount the on-line UNIX System manual entries from a specified server. The automounter also allows a client to configure redundant servers. That is, it allows a client to configure a set of servers from which a particular file system can be mounted. Therefore, in this example, instead of just specifying `bigmo` as the remote server for `/usr/man`, it would be possible to specify remote servers `bigjoe` and `bigben`, as well. If configured in this way, the automounter would locate the first available server (using an RPC `ping` operation) and then NFS mount `/usr/man` from that system.

8.2 Using the Automounter

8.2.1 Preparing the Maps

Unlike the `mount` command, `automount` does not consult `/etc/fstab` for information on which file systems to mount. Rather, it consults the map file(s) specified on the command line at startup time (see section 8.2.2, "Starting Automount"). If no maps are specified, it looks for an NIS map called `auto.master`. If no NIS `auto.master` exists, `automount` exits silently.

By convention, all automounter maps are located in the directory `/etc` and have file names prefixed with `auto.`

There are three kinds of automount maps:

1. master
2. indirect
3. direct

The master map lists (as if from the command line) all other maps, applicable options, and mount points as described below and summarized in the following figure:

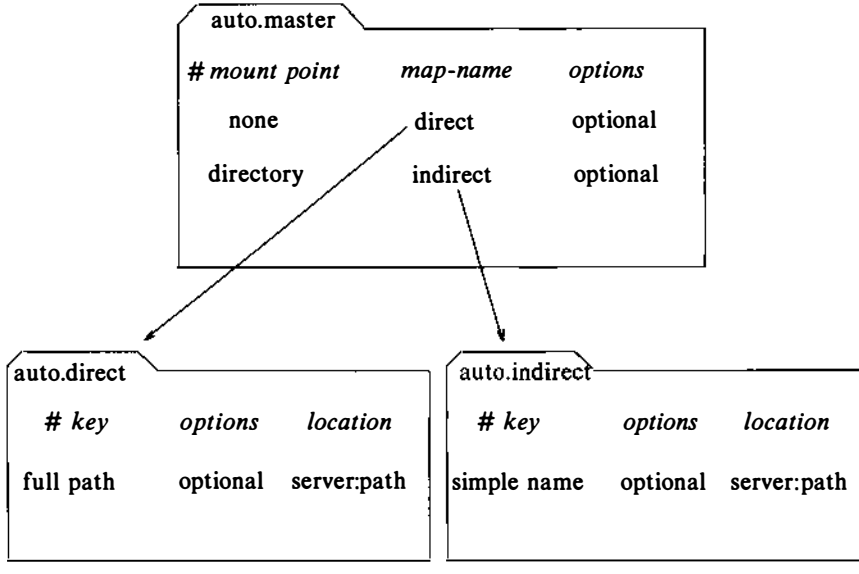


Figure 6. Master, Direct, and Indirect Maps

8.2.1.1 The Master Map. Each line in the master map, by convention called `/etc/auto.master`, has the syntax:

mount-point map-name [*mount-options*]

where:

- *mount-point* is the full path name of a directory. If the directory does not exist, the automounter will create it if possible. If the directory exists and is not empty, mounting on it will hide its contents. In this case, the automounter will issue a warning message.
- *map-name* is the map the automounter should use to find the mount points and locations.
- *mount-options* is an optional, comma-separated list of options that regulate the mounting of the entries mentioned in *map-name*, unless the entries in *map-name* list other options.

A line whose first character is a # is treated as a comment, and everything that follows until the end of line is ignored. A backslash (\) at the end of a line permits splitting long lines into shorter ones. The notation /- as a mount point indicates that the map in question is a direct map and no particular mount point is associated with the map as a whole.

8.2.1.2 Direct and Indirect Maps. Lines in direct and indirect maps have the syntax:

key [*mount-options*] *location*

where:

- *key* is the path name of the mount point.
- The *mount-options* are the options you want to apply to this particular mount.
- *location* is the location of the resource, specified as *server:pathname*[*:subdirectory*].

As in the master map, a line whose first character is a # is treated as a comment and everything that follows until the end of line is ignored. A backslash at the end of line permits splitting long lines into shorter ones.

The only formal difference between a direct and an indirect map is that the key in a direct map is a full path name, whereas in an indirect path name it is a simple name (no slashes). For instance, the following would be an entry in a direct map:

```
/usr/man        -ro     goofy:/usr/man
```

and the following would be an entry in an indirect map:

```
parsley         -ro     veggies:/usr/greens
```

Clearly, the *key* in the indirect map requires more information, specifically the actual location of the mount point, `parsley`. This information must be provided at the command line or through another map. For instance, if the above line is part of a map called `/etc/auto.veggies`, the appropriate level of qualification can be provided through invoking the `automount` command as follows:

```
# automount /veggies /etc/auto.veggies
```

or adding the following specification to the master map:

```
/veggies        /etc/auto.veggies        -ro, soft, nosuid
```

In either case, the mount directory, `/veggies`, is being used to qualify the entries in the indirect map `/etc/auto.veggies`. The end result is that the file system `/usr/greens` from the machine `veggies` will be mounted on `/veggies/parsley`, when needed.

8.2.1.3 Writing a Master Map. As stated above, the syntax for each line in the master map is:

mount-point map-name [mount-options]

A typical `auto.master` file would contain:

```
/-      /etc/auto.direct      -ro
/home   /etc/auto.home        -rw
/net    -hosts
```

The automounter recognizes some special mount points and maps, which are explained below.

Mount point `/-`

In the example above, the mount point `/-` is a filler that the automounter recognizes as a directive not to associate the entries in `/etc/auto.direct` with any directory. Rather, the mount points are to be the ones mentioned in the map. (Remember, in a direct map the key is a full path name.)

Mount point `/home`

The mount point `/home` is to be the directory under which the entries listed in `/etc/auto.home` (an indirect map) are to be mounted. That is, they will be mounted under `/tmp_mnt/home`, and the NFS symbolic link machinery will be used to associate `/home/directory` and `/tmp_mnt/home/directory`.

Mount point `/net`

Finally, the automounter will mount under the directory `/net` all the entries under the special map, `-hosts`. This is a built-in map that does not use any external files except the hosts database `/etc/hosts` or the NIS map, `hosts.byname`, if NIS is running. Notice that since the automounter does not mount the entries until needed, the specific order is not important. Once the `automount` daemon is in place, a user entering the command:

```
$ cd /net/gumbo
```

will change directory to the top of the file system (i.e., the `root` file system) of the machine `gumbo`, as long as the machine is in the hosts database and it exports any of its file systems. However, the user may not see all of the files and directories under `/net/gumbo` because the automounter can mount only the exported file systems of host `gumbo`, in accordance with the restrictions placed on the exporting.

The actions of the automounter when the command in the example above is issued are as follows:

1. `ping` the null procedure of the server's mount service to see if it is alive.
2. Request the list of exported file system from the server.
3. Sort the exported list according to the length of the path name.

   ```
   /usr/src
   /export/home
   /usr/src/sccs
   /export/root/blah
   ```

   This sorting ensures that the mounting is done in the proper order, that is, `/usr/src` is done before `/usr/src/sccs`.
4. Proceed down the list, mounting all the file systems at mount points in `/tmp_mnt` (creating the mount points as needed).

Note that the automounter has to mount all of the file systems that the server in question exports. Therefore, if the user issues the following `ls` command:

```
$ ls /net/gumbo/usr/include
```

the automounter mounts all of `gumbo`'s exported systems, not just `/usr`.

In addition, unmounting that occurs after a certain amount of time has passed is from the bottom up. This means that if one of the directories at the top is busy, the automounter has to remount the file system and try again later.

Nevertheless, the `-hosts` special map provides a very convenient way for users to access directories on many different hosts without having to use the `rlogin` or `rsh` command.

In addition, users no longer have to modify their `/etc/fstab` files or mount the directories by hand as superuser.

Notice that both `/net` and `/home` are arbitrary names dictated by convention. The automounter will create them if they do not exist already.

8.2.1.4 Writing an Indirect Map. The syntax for an indirect map is:

key [*mount-options*] *location*

where *key* is the basename (not the full path name) of the directory that will be used as mount point. Once the key is obtained by the automounter, it is suffixed to the mount point associated with it either by the command line or by the master map that invokes the indirect map in question.

For instance, one of the entries in the master map presented above as an example reads:

```
/home   /etc/auto.home  -rw
```

Here `/etc/auto.home` is the name of the indirect map that will contain the entries to be mounted under `/home`.

A typical `auto.home` map might contain:

```
willow                  willow:/home/willow
cypress                 cypress:/home/cypress
poplar                  poplar:/home/poplar
pine                    pine:/export/pine
apple                   apple:/export/home
ivy                     ivy:/home/ivy
peach   -rw,nosuid      peach:/export/home
```

As an example, assume that the map above is on the host `oak`. If the user `laura` has an entry in the password database specifying her home directory as `/home/willow/laura`, whenever she logs into machine `oak`, the automounter will mount (as `/tmp_mnt/home/willow`) the directory `/home/willow` residing on machine `willow`. If one of the directories is indeed `laura`, she will be in her home directory, which is mounted `read/write` as specified by the options field in the master map entry.

Suppose, however, that `laura`'s home directory is specified as `/home/peach/laura`. Whenever she logs into `oak` the automounter mounts the directory `/export/home` from `peach` under `/tmp_mnt/home/peach`. Her home directory will be

mounted `read/write, nosuid`. Any option in the file entry overrides all options in the master map or the command line.

Now, assume the following conditions occur:

- User `laura`'s home directory is listed in the password database as `/home/willow/laura`.
- The machine, `willow`, exports its `home` file system to the machines mentioned in `auto.home`.
- All those machines have a copy of the same `auto.home` and the same password database.

Under these conditions, the user, `laura`, can run the `login` command or `rlogin` on any of these machines and have her home directory mounted in place for her.

In addition, `laura` can also enter the command:

```
$ cd ~bigmo
```

and the automounter will mount `bigmo`'s home directory for her (if all permissions apply).

In order to accomplish this on a network without NIS, you must change all of the relevant databases (such as `/etc/passwd`) on all of the systems on the network. On a network running NIS, make the changes on the NIS master server and propagate the relevant databases to the slave servers.

8.2.1.5 Writing a Direct Map. The syntax for a direct map (like that for an indirect map) is:

key [*mount-options*] *location*

where:

- *key* is the *full* path name of the mount point. (Remember that in an indirect map this is not a full path name.)
- *mount-options* are optional but, if present, override the options of the calling line or the defaults for the entry in question. (See section 8.2.2, "Starting Automount.")
- *location* is the location of the resource, specified as *server:pathname* [*:subdirectory*].

Of all the maps, the entries in a direct map most closely resemble, in their simplest form, what their corresponding entries in

`/etc/fstab` might look like. An entry that appears in `/etc/fstab` as:

```
dancer:/usr/local/bin  /usr/local/bin  -r NFS
```

appears in a direct map as:

```
/usr/local/bin          -ro             dancer:/usr/local/bin
```

The following is a sample `/etc/auto.direct` map:

```
/usr/local \
              /bin      -ro,soft        ivy:/usr/local/bin \
              /src      -ro,soft        ivy:/usr/local/src
/usr/man                -ro,soft        oak:/usr/man \
                                        rose:/usr/man \
                                        willow:/usr/man
/usr/games              -ro,soft        peach:/usr/games
/usr/spool/news         -ro,soft        pine:/usr/spool/news
```

There are two unusual features in this map: multiple mounts and multiple locations. These are the subject of the next two sections.

8.2.1.6 Multiple Mounts. A map entry can describe a multiplicity of mounts, where the mounts can be from different locations and with different mount options. Consider the first entry in the previous example:

```
/usr/local \
              /bin      -ro,soft        ivy:/usr/local/bin \
              /src      -ro,soft        ivy:/usr/local/src
```

This is, in fact, one long entry whose readability has been improved by splitting it into three lines using the backslash and indenting the continuation lines with blank spaces or tabs. This entry mounts `/usr/local/bin` and `/usr/local/src` from the server `ivy`, with the options `read-only` and `soft`. The entry could also read:

```
/usr/local \
               /bin     -ro,soft        ivy:/usr/local/bin \
               /src     -ro             oak:/usr/local/src
```

where the options are different and more than one server is used. The difference between the above and two separate entries, for example:

```
/usr/local/bin          -ro,soft        ivy:/usr/local/bin
/usr/local/src          -ro             oak:/usr/local/src
```

is that the first case, a multiple mount, guarantees that both directories will be mounted when you reference one of them. In the case of the separate entries, if you, for instance, enter:

```
$ cd /usr/local/bin
```

you cannot `cd` to the other directory using a relative path, because it is not mounted yet:

```
$ cd ../src
../src: No such file or directory
```

A multiple mount obviates this problem. In multiple mounts, each file system is mounted on a subdirectory within another file system. When the `root` of the file system is referenced, the automounter mounts the whole file system.

A special case of multiple mounts occurs when the `root` of a file system has to be mounted as well. This is called a *hierarchical* mount. The following illustration shows a true hierarchical mounting:

```
/usr/local \
                /       -rw         peach:/usr/local \
                /bin    -ro,soft    ivy:/usr/local/bin \
                /src    -ro         oak:/usr/local/src
```

Note that a true hierarchical mount can be problematic if the server for the `root` of the file system goes down. Any attempt to unmount the lower branches will fail, since the unmounting has to proceed through the mount `root`, which also cannot be unmounted while its server is down.

Finally, a word about mount options. In one of the examples above:

```
/usr/local \
                 /bin    -ro,soft      ivy:/usr/local/bin \
                 /src    -ro,soft      oak:/usr/local/src
```

both mounts share the same options. This could be modified to:

```
/usr/local              -ro,soft \
                /bin                  ivy:/usr/local/bin \
                /src                  oak:/usr/local/src
```

If one of the mount points needed a different specification, you could then write:

```
/usr/local              -ro,soft \
                /bin                  ivy:/usr/local/bin \
                /src    -rw           oak:/usr/local/src
```

8.2.1.7 Multiple Locations. In the example for a direct map, which was:

```
/usr/local \
                /bin    -ro,soft        ivy:/usr/local/bin \
                /src    -ro,soft        ivy:/usr/local/src
/usr/man                -ro,soft        oak:/usr/man \
                                        rose:/usr/man \
                                        willow:/usr/man
/usr/games              -ro,soft        peach:/usr/games
/usr/spool/news         -ro,soft        pine:/usr/spool/news
```

the mount point `/usr/man` lists more than one location. This means that the mounting can be done from any of the replicated locations, in this case, `oak`, `rose`, or `willow`. This list can also be expressed as a comma-separated list of servers, followed by the colon and the path name (as long as the path name is the same for all of the replicated servers), for example:

```
/usr/man        -ro,soft        oak,rose,willow:/usr/man
```

The first server to respond to the RPC `ping` issued by the automounter is selected, and an attempt is made to mount from it. Note that the list does not imply an ordering, even though servers on the local network will be pinged first.

This redundancy, which is very useful in an environment where individual servers may or may not be exporting their file systems, is enjoyed only at mount time. There is no status checking of the mounted-from server by the automounter once the mount occurs. If the server goes down while the mount is in effect, the file system becomes unavailable. One option is to wait 5 minutes until the auto-unmount takes place and try again. Next time around the automounter will choose one of the other, available servers. Another option is to use the `umount` command, inform the automounter of the change in the mount table (as specified in section 8.2.2.1, "The Mount Table"), and retry the mount.

Note that care should be taken when using multiple locations for `read/write` file systems, since the actual server may change from access to access.

8.2.1.8 Specifying Subdirectories. Section 8.2.1.4, "Writing an Indirect Map," showed the following typical `auto.home` file:

```
willow                  willow:/home/willow
cypress                 cypress:/home/cypress
poplar                  poplar:/home/poplar
pine                    pine:/export/pine
apple                   apple:/export/home
ivy                     ivy:/home/ivy
peach   -rw,nosuid      peach:/export/home
```

Given this `auto.home` indirect file, every time a user wants to access a home directory in, for example, `/home/willow`, all of the directories under it will be mounted. Another way to organize an `auto.home` file is by user name, as in:

```
john            willow:/home/willow/john
mary            willow:/home/willow/mary
joe             willow:/home/willow/joe
```

The above example assumes that home directories are of the form `/home/user` rather than `/home/server/user`. If a user now enters the following command:

```
$ ls ~john ~mary
```

the automounter has to perform the *equivalent* of the following actions:

```
# mkdir /tmp_mnt/home/john
# mount -f NFS willow:/home/willow/john  /tmp_mnt/home/john
# ln -s /tmp_mnt/home/john  /home/john

# mkdir /tmp_mnt/home/mary
# mount -f NFS willow:/home/willow/mary  /tmp_mnt/home/mary
# ln -s /tmp_mnt/home/mary  /home/mary
```

It is possible to optimize the work done by the automounter, by modifying the entries in the `auto.home` map to use the optional *subdirectory* field of the *location.* The new `auto.home` map would look like:

```
john            willow:/home/willow:john
mary            willow:/home/willow:mary
joe             willow:/home/willow:joe
```

Here `john`, `mary`, and `joe` are entries in the *subdirectory* field. Now, when a user refers to `john`'s home directory, the automounter mounts `willow:/home/willow` and links `/tmp_mnt/home/willow/john` and `/home/john`.

If the user then requests access to `mary`'s home directory, the automounter sees that `willow:/home/willow` is already mounted and simply links `/tmp_mnt/home/willow/mary` and `/home/mary`. In other words, the automounter now only performs the *equivalent* of the following:

```
# mkdir /tmp_mnt/home/john
# mount -f NFS willow:/home/willow  /tmp_mnt/home
# ln -s /tmp_mnt/home/john  /home/john
# ln -s /tmp_mnt/home/mary /home/mary
```

In general, it is a good idea to provide a *subdirectory* entry in the `location` when different map entries refer to the same mounted file system from the same server.

8.2.1.9 Metacharacters. The automounter recognizes some characters as having a special meaning. Some are used for substitutions, some to escape other characters.

Ampersand (&)

If you have a map with many subdirectories specified, for example:

```
john            willow:/home/willow:john
mary            willow:/home/willow:mary
joe             willow:/home/willow:joe
able            pine:/export/home:able
baker           peach:/export/home:baker
      [. . .]
```

consider using string substitutions. You can use the ampersand character (&) to substitute the key wherever it appears.

Using the ampersand, the above map now looks as follows:

```
john            willow:/home/willow:&
mary            willow:/home/willow:&
joe             willow:/home/willow:&
able            pine:/export/home:&
baker           peach:/export/home:&
      [. . .]
```

If the name of the server is the same as the key itself, for instance:

```
willow          willow:/home/willow
peach           peach:/home/peach
pine            pine:/home/pine
oak             oak:/home/oak
poplar          poplar:/home/poplar
      [. . .]
```

the use of the ampersand results in:

```
willow          &:/home/&
peach           &:/home/&
pine            &:/home/&
oak             &:/home/&
poplar          &:/home/&
      [. . .]
```

Asterisk (*)

Notice that all of the above entries have the same format. This permits you to use the catch-all substitute character, the asterisk (*). The asterisk reduces the whole thing to:

```
*       &:/home/&
```

where each ampersand is substituted by the value of any given key. Notice that once the automounter reads the catch-all

key, it does not continue reading the map, so that the following map would be viable:

```
oak             &:/export/&
poplar          &:/export/&
*               &:/home/&
```

but in the next map the last two entries would always be ignored:

```
*               &:/home/&
oak             &:/export/&
poplar          &:/export/&
```

You could also use key substitutions in a direct map, in situations like the following:

```
/usr/man                    willow,cedar,poplar:/usr/man
```

which is a good candidate to be written as:

```
/usr/man                    willow,cedar,poplar:&
```

Notice that the ampersand substitution uses the whole key string, so if the key in a direct map starts with a / (as it should), that slash is carried over, and you could not do something like:

```
/progs          &1,&2,&3:/export/src/progs
```

because the automounter would interpret it as:

```
/progs          /progs1,/progs2,/progs3:/export/src/progs
```

Backslash (\)

Under certain circumstances you may have to mount directories whose names may confuse the automounter's map parser. This is a concern only with non-UNIX System servers. An example might be a directory called `rc0:dk1`; this could result in an entry like:

```
/junk   -ro     vmsserver:rc0:dk1
```

The presence of the two colons in the `location` field will confuse the automounter's parser. To avoid this confusion, use a backslash to escape the second colon and remove its special meaning of separator:

```
/junk   -ro     vmsserver:rc0\:dk1
```

Double quotes (")

You can also use double quotes, as in the following example, where they are used to hide the blank space in the name:

```
/smile          dentist:/"front teeth"/smile
```

8.2.1.10 Environment Variables. Environmental variables can be used by prefixing a dollar sign (`$`) to the name of the variable. Braces are used to delimit the name of the environmental variable from appended letters or digits. These variables can be used anywhere in an entry line, except as a *key*.

Environmental variables can either be inherited from the environment or can be defined explicitly with the `-D` command line option. For instance, if you want each client to mount client-specific files in the network in a replicated format, you could create a specific map for each client according to its name, so that the relevant line for host `oak` would be:

```
/mystuff          cypress,ivy,balsa:/export/hostfiles/oak
```

and for `willow` it would be:

```
/mystuff          cypress,ivy,balsa:/export/hostfiles/willow
```

This scheme is viable within a small network, but maintaining this kind of host-specific map across a large network would soon become unfeasible. The solution in this case would be to start the automounter with a command line similar to the following:

```
# automount -D HOST=`hostname` ...
```

and have the entry in the direct map read:

```
/mystuff          cypress,ivy,balsa:/export/hostfiles/$HOST
```

Now each host would find its own files in the `mystuff` directory, and the task of centrally administering and distributing the maps becomes easier.

8.2.1.11 Including Other Maps. A line of the form, *+mapname*, causes the automounter to consult the mentioned map as if it were included in the current map. If *mapname* is a relative path name (no slashes), the automounter assumes it is an NIS map. If the path name is an absolute path name, the automounter looks for a local map of that name. If the *mapname* starts with a dash (`-`), the automounter consults the appropriate built-in map.

For instance, you can have a few entries in your local `auto.home` map for the most commonly accessed home directories and follow them with the included NIS map:

```
ivy     -rw     &:/home/&
oak     -rw     &:/export/home
+auto.home
```

After consulting the included map, the automounter continues scanning the current map if no match is found, so you can add more entries. For example:

```
ivy     -rw     &:/home/&
oak     -rw     &:/export/home
+auto.home
*       -rw     &:/home/&
```

Finally, as mentioned before, the map included can be a local file or even a built-in map:

```
+auto.home.finance                  # NIS map
+auto.home.sales                    # NIS map
+auto.home.engineering              # NIS map
+/etc/auto.mystuff                  # local map
+auto.home                          # NIS map
+-hosts                             # built-in hosts map
*               &:/export/&         # wild card
```

8.2.2 Starting Automount

Once the maps are written, you should make sure that there are no equivalent entries in `/etc/fstab` and that all the entries in the maps refer to NFS exported files.

The syntax to invoke the automounter is:

```
automount [-mnTv] [-D name = value] [-f master-file] [-M mount-directory]
[-tl duration] [-tm interval] [-tw interval] [directory map [-mount-options] ] ...
```

The *automount*(1M) manual entry contains a complete description of all options. The mount options that you can specify on the command line or in the maps are the same as those for a standard NFS `mount`, excluding `bg` (background) and `fg` (foreground), which do not apply.

By default, if the `/etc/auto.master` file exists, the NFS startup script, `/etc/rc3.d/s72nfs`, starts the automounter at boot time through the lines:

```
echo "automount\c"
automount -m -f /etc/auto.master
```

The `-m` option instructs the automounter not to look for the NIS map; the `-f` option instructs it to use the local file, `/etc/auto.master`, as the master map.

In order to use the NIS `auto.master` map, `automount` should be started without the `-m` option:

```
# automount
```

If the map is found, the automounter follows the directives contained within. If NIS is not running or the map is not found, the automounter exists silently.

It is also possible to specify master map information, that is, mount points, map names, and mount options on the `automount` command line itself:

```
# automount /net -hosts /home /etc/auto.home -rw /- /etc/auto.direct -ro
```

This is equivalent to starting `automount` with no options and the following `/etc/auto.master` map:

```
/net    -hosts
/home   /etc/auto.home      -rw
/-      /etc/auto.direct    -ro
```

Other combinations of `automount` arguments can be used to change the name of the master map and add, nullify, or override master map entries. Again, see the *automount*(1M) manual entry for more details.

8.2.2.1 The Mount Table. Every time the automounter mounts or unmounts a file system, it modifies `/etc/mnttab` to reflect the current situation. The automounter keeps an image in memory of `/etc/mnttab` and refreshes this image every time it performs a mounting or an automatic unmounting. If you use the `umount` command to unmount one of the automounted file systems (a directory under `/tmp_mnt`), the automounter should be forced to re-read the `/etc/mnttab` file. To do that, enter the following command:

```
# ps -ef | grep automount | egrep -v grep
```

This gives you the process ID of the automounter. The automounter is designed so that on receiving a SIGHUP signal it re-reads `/etc/mnttab`. In order to send it that signal, enter:

```
# kill -1 PID
```

where *PID* stands for the process ID you obtained from the previous `ps` command.

8.2.2.2 Modifying the Maps. You can modify the automounter maps at any time, but that does not guarantee that all of your modifications will take effect the next time the automounter mounts a file system. It depends on what map you modify and what kind of modification you introduce. You may have to reboot the machine.

This is generally the simplest way of restarting the automounter, although, if it is used sparingly, you could theoretically kill it and restart it from the command line.

8.2.2.3 Modifying the Master Map. The automounter consults the master map only at startup time. A modification to the master map will only take effect the next time you reboot the machine.

8.2.2.4 Modifying Indirect Maps. Entries can be modified, deleted, or added to indirect maps, and the change will take effect the next time the map is used, which is the next time a mount has to be done.

8.2.2.5 Modifying Direct Maps. Each entry in a direct map is an automount mount point and it only mounts itself at these mount points at startup. Therefore, adding or deleting an entry in a direct map will only take effect the next time you reboot the machine. However, existing entries can be modified (mount options or server names can be changed, for example, but not the names of mount points) while the automounter is running and will take effect when the entry is next mounted because the automounter consults the direct maps whenever a mount has to be done.

For example, if you modify the file `/etc/auto.direct` so that the directory `/usr/src` is now mounted from a different server, the new entry takes effect immediately (if `/usr/src` is not mounted at this time) when you try to access it. If it is mounted at this time, you can wait until the auto unmounting takes place and then access it. If this is not satisfactory, you can unmount with the `umount` command, notify `automount` that the mount table has changed (see section 8.2.2.1, "The Mount Table"), and then access it. The mounting should now be done from the new server. Note, however, that if you wanted to delete the entry, you would have to reboot the machine for the deletion to take effect.

For this reason and because they do not clutter the mount table like direct maps do, indirect maps are preferable and should be used whenever possible.

8.2.2.6 Mount Point Conflicts. If you have a home partition on a local disk that is mounted on `/home` and you also want to use the automounter to mount other home directories, you may find that if you specify the mount point `/home`, the automounter will hide the local home partition whenever you try to reach it.

The solution is to mount the partition somewhere else, for example, on `/export/home`. You would then need, for example, an entry in `/etc/fstab` that specifies:

```
/dev/xx#z       ...       /export/home    ...
```

(where *xx#z* stands for the name of the partition). The master file must contain a line similar to this:

```
/home           /etc/auto.home
```

and there must be an entry in `auto.home` that specifies:

```
terra           terra:/export/home
```

where `terra` is the name of the machine.

If the partition is set up such that home directories are to be found in `/home/machine/user`, move all the directories at the `user` level one level up to eliminate the `machine` level:

```
# cd /home
# mv machine/* .
# rmdir machine
```

There is no need to change the `/etc/passwd` entry for the user. The user's home directory will still be accessible through `/home/machine/user`, as before. Instead of doing a mount, the automounter will recognize that the file system is on the same machine and will establish a symbolic link from `/home/machine` to `/export/home`.

8.3 Error Messages

The following paragraphs are error messages you are likely to see if the automounter fails, and an indication of what the problem may be. Note that all automounter error messages are prefixed by the string "`automount:`".

8.3.1 Error Messages Generated by the Verbose Option

`no mount maps specified`
: The automounter was invoked with no maps to serve, and it cannot find the NIS `auto.master` map. It exits. Recheck the command, or restart NIS if that was the intention.

mapname: `not found`
: The required map cannot be located. This message is produced only when the `-v` option is given. Check the spelling and path name of the map name.

`leading space in map entry` *entry text*
`in` *mapname*
: The automounter has discovered an entry in an automount map that contains leading spaces. This is usually an indication of an improperly continued map entry, for example:

```
foo
        /bar    frobz:/usr/frotz
```

In the example above, the warning is generated when the automounter encounters the second line, because the first line should be terminated with a backslash (\).

`bad key` *'key'* `in indirect map` *mapname*
: While scanning an indirect map, the automounter has found an entry key containing a /. Indirect map keys must be simple names, not path names.

`bad key` *'key'* `in direct map` *mapname*
: While scanning a direct map the automounter has found an entry key without a prepended /. Keys in direct maps must be full path names.

`NIS bind failed`
: The automounter was unable to communicate with the `ypbind` daemon. This is information only; the automounter will continue to function correctly provided it requires no explicit NIS support. If you need NIS, check to see whether there is a `ypbind` daemon running.

`couldn't create mntpnt` *'mountpoint'*`:` *reason*
: The automounter was unable to create a mountpoint required for a mount. This usually happens when attempting to hierarchically mount all of a server's exported file systems. A required mountpoint may exist only in a file system that cannot be mounted (it may not be exported), and it cannot be created because the exported parent file system is exported `read only`.

`WARNING:` *mountpoint* `already mounted on`
: The automounter is attempting to mount over an existing mountpoint. This is indicative of an internal error in the automounter (a bug).

server:pathname `already mounted on` *mountpoint*
: The automounter is attempting to mount over a previous mount of the same file system. This could happen if an entry

appears both in `/etc/fstab` and in an automounter map (either by accident or because the output of `mount -p` was redirected to `fstab`). Delete one of the redundant entries.

`can't mount`*server:pathname: reason*
The mount daemon on the server refuses to provide a file handle for *server:pathname*. Check the export table on *server*.

`remount` *server:pathname* `on` *mountpoint*`:`
`server not responding`
The automounter has failed to remount a file system it previously unmounted. This message may appear at intervals until the file system is successfully remounted.

`WARNING:` *mountpoint* `not empty!`
The mount point is not an empty directory. The directory *mountpoint* contains entries that will be hidden while the automounter is mounted there. This is advisory only.

8.3.2 General Error Messages

`pathok: couldn't find devid` *device id*
An internal automounter error (bug).

`WARNING: default option` *'option'*
`ignored for map` *mapname*
Where *option* is an unrecognized default mount option for the map *mapname*.

`WARNING:` *option* `ignored for` *key* `in` *mapname*
The automounter has detected an unknown mount option. This is advisory only. Correct the entry in the appropriate map.

`bad entry in map` *mapname key*
`map` *mapname*`, key` *key*`: bad`
The map entry is malformed and the automounter cannot interpret it. Recheck the entry; there may be characters in it that must be escaped.

`can't get my address`
The automounter cannot find an entry for its host in `/etc/hosts` (or `hosts.byname`).

`cannot create UDP service`
The automounter cannot establish a UDP connection.

`svc_register failed`
: Automounter cannot register itself as an NFS server. Check the kernel configuration file.

`couldn't create` *pathname: reason*
: Where *pathname* is `/tmp_mnt` or the argument to the `-M` command line option.

`can't mount` *mountpoint: reason*
`nmount may need increasing`
: The automounter couldn't mount its daemon at *mountpoint*.

`exiting`
: This is an advisory message only. The automounter has received a SIGTERM (has been killed) and is exiting.

server:pathname `no longer mounted`
: The automounter is acknowledging that *server:pathname* which it mounted earlier has been unmounted by the `umount` command. The automounter will notice this within 1 minute of the unmount or immediately if it receives a SIGHUP.

`trymany: servers not responding:` *reason*
: No server in a replicated list is responding. This may indicate a network problem.

`host` *server* `not responding`
: The automounter attempted to contact *server* but received no response.

`mount of` *server:pathname* `on` *mountpoint: reason*
: The automounter failed to do a mount. This may indicate a server or network problem.

`hierarchical mountpoints:` *pathname1*
`and` *pathname2*
: The automounter does not allow its mount points to have a hierarchical relationship, that is, an automounter mount point must not be contained within another automounted file system.

mountpoint: `Not a directory`
: The automounter cannot mount itself on *mountpoint* because it is not a directory. Check the spelling and path name of the mount point.

`dir` *mountpoint* `must start with '/'`
: The automounter mount point must be given as a full path name. Check the spelling and path name of the mount point.

mapname: yp_err
: Error in looking up an entry in an NIS map. This may indicate NIS problems.

hostname: `exports:` *rpc_err*
: Error getting export list from *hostname*. This indicates a server or network problem.

`nfscast: Cannot send packet:` *reason*
: The automounter cannot send a query packet to a server in a list of replicated file system locations.

`nfscast: cannot receive reply:` *reason*
: The automounter cannot receive replies from any of the servers in a list of replicated file system locations.

`nfscast:select:` *reason*
`Cannot create socket for nfs:` *rpc_err*
: These error messages indicate problems attempting to `ping` servers for a replicated file system. This may indicate a network problem.

INTERACTIVE NFS
User's Reference Manual

CONTENTS

NAME

intro – introduction to Open Network Computing (ONC) commands

DESCRIPTION

This section describes publicly accessible ONC utilities in alphabetical order.

SEE ALSO

"INTERACTIVE NFS Administrator's Reference Manual" (the Section 1M manual entries) for ONC administration commands.
exit(2), wait(2) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

DIAGNOSTICS

Upon termination, each command returns 2 bytes of status data: one supplied by the system giving the cause for termination and (in the case of "normal" termination) one supplied by the program (see *wait*(2) and *exit*(2)). The former byte is 0 for normal termination. The latter is customarily 0 for successful execution; a nonzero value indicates troubles such as erroneous parameters, bad or inaccessible data, or some other inability to cope with the task at hand. This datum is called variously "exit code," "exit status," or "return code"; it is described only where special conventions are involved.

NAME

on – execute a command remotely

SYNOPSIS

on [**–i**] [**–n**] [**–d**] host command [argument] ...

DESCRIPTION

The *on* program is used to execute commands on another system in an environment similar to that invoking the program. All environment variables are passed, and the current working directory is preserved. To preserve the working directory, the working file system must be either already mounted on the host or exported to it. Relative path names will only work if they are within the current file system; absolute path names may cause problems.

Standard input is connected to standard input of the remote command, and standard output and standard error from the remote command are sent to the corresponding files for the *on* command.

The *on* command has the following options:

–i Interactive mode. Use remote echoing and special character processing. This option is needed for programs that expect to be talking to a terminal. All terminal modes and window size changes are propagated.

–n No input. This option causes the remote program to get end-of-file when it reads from standard input instead of passing standard input from the standard input of the *on* program. For example, **–n** is necessary when running commands in the background with job control.

–d Debug mode. Print out some messages as work is being done.

SEE ALSO

rexd(1M), exports(4).

DIAGNOSTICS

unknown host	Host name not found
cannot connect to server	Host down or not running the server
can't find .	Problem finding the working directory
can't locate mount point	Problem finding current file system

Other error messages may be passed back from the server.

NAME

rpcgen − an RPC protocol compiler

SYNOPSIS

rpcgen infile
rpcgen −h [**−o** outfile] [inputfile]
rpcgen −c [**−o** outfile] [infile]
rpcgen [**−s** transport]* [**−o** outfile] [infile]
rpcgen −l [**−o** outfile] [infile]
rpcgen −m [**−o** outfile] [infile]

DESCRIPTION

The *rpcgen* compiler is a tool that generates C code to implement an RPC protocol. The input to *rpcgen* is a language similar to C, known as RPC Language (Remote Procedure Call Language).

The *rpcgen* command is normally used as shown in the first synopsis, where it takes an input file and generates four output files. If the *infile* is named *proto*.**x**, then *rpcgen* generates a header file in *proto*.**h**, XDR routines in *proto_xdr*.**c**, server-side stubs in *proto_svc*.**c**, and client-side stubs in *proto_clnt*.**c**.

The other synopses shown above are used when one wants to generate a particular output file rather than all the files.

Since the C preprocessor, *cpp*(1), is run on all input files before they are actually interpreted by *rpcgen*, all the *cpp* directives are legal within an *rpcgen* input file. For each type of output file, *rpcgen* defines a special *cpp* symbol for use by the *rpcgen* programmer:

RPC_HDR Defined when compiling into header files

RPC_XDR Defined when compiling into XDR routines

RPC_SVC Defined when compiling into server-side stubs

RPC_CLNT Defined when compiling into client-side stubs

In addition, *rpcgen* does a little preprocessing of its own. Any line beginning with "%" is passed directly into the output file, uninterpreted by *rpcgen*.

You can customize some of your XDR routines by leaving those data types undefined. For every data type that is undefined, *rpcgen* assumes that there exists a routine with the name **xdr_**prepended to the name of the undefined type.

The following options are available:

−c Compile XDR routines.

−h Compile into **C** data definitions (a header file).

−l Compile into client-side stubs.

−s *transport*
Compile server-side stubs using the given transport. The supported transports are **udp** and **tcp**. This option may be invoked more than once so as to compile a server that serves multiple transports.

−m Compile into server-side stubs, but do not produce a *main*() routine. This option is useful if you want to supply your own *main*().

−o *outfile*
Specify the name of the output file. If none is specified, standard output is used (**−c**, **−h**, **−l**, and **−s** modes only).

SEE ALSO

The section "RPCGEN PROTOCOL COMPILER" in the "INTERACTIVE NFS Protocol Specifications and User's Guide."

BUGS

Nesting is not supported. As a workaround, structures can be declared at top level and the name of each used inside other structures in order to achieve the same effect.

Name clashes can occur when using program definitions, since the apparent scoping does not really apply. Most of these can be avoided by giving unique names to programs, versions, procedures, and types.

INTERACTIVE NFS
Programmer's Reference Manual

CONTENTS

NAME

intro – introduction to NFS system calls and error numbers

SYNOPSIS

#include <sys/errno.h>

DESCRIPTION

This section describes all of the socket system calls used in System V NFS. Some of these system call are accessible from the RPC library, *librpc*. The rest of the system calls were designed for specific purposes for specific programs. The system call interfaces are generally built into these programs. There are no new error numbers added for the support of the NFS system calls. Some of these system calls were not designed to return during normal operation. They were designed to give the kernel a user context to run in or to provide the kernel with a resource that is more easily allocated from the user level. Most of these calls have one or more error returns. An error condition is indicated by an otherwise impossible return value. This is almost always −1; the individual descriptions specify the details.

As with normal arguments, all return codes and values from functions are of type integer unless noted otherwise. An error number is also made available in the external variable *errno*, which is not cleared on successful calls. Thus, *errno* should be tested only after an error has occurred.

See *intro*(2) for the standard error codes.

List of Functions

Name	*Appears on Entry*	*Description*
getdomainname	getdomainname(2)	return the NIS domain name
setdomainname	getdomainname(2)	set the NIS domain name

FILES

/usr/lib/librpc.a

SEE ALSO

intro(2), perror(3C) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual*.

NAME

getdomainname, setdomainname – get/set name of current domain

SYNOPSIS

```
getdomainname(name, namelen)
char *name;
int namelen;

setdomainname(name, namelen)
char *name;
int namelen;
```

DESCRIPTION

The *getdomainname* call returns the name of the domain for the current processor, as previously set by *setdomainname*. The parameter *namelen* specifies the size of the *name* array. The returned name is null-terminated unless insufficient space is provided.

The *setdomainname* call sets the domain of the host machine to be *name*, which has length *namelen*. This call is restricted to the superuser and is normally used only when the system is bootstrapped.

The purpose of domains is to enable two distinct networks that may have host names in common to be merged. Each network would be distinguished by having a different domain name. At the current time, only the network information service makes use of domains.

RETURN VALUES

If the call succeeds, a value of 0 is returned. If the call fails, a value of −1 is returned, and an error code is placed in the global location *errno*.

ERRORS

The following errors may be returned by these calls:

[EFAULT]	The *name* parameter gave an invalid address.
[EPERM]	The caller was not the superuser. This error only applies to *setdomainname*.

BUGS

Domain names are limited to 64 characters.

NAME

mount – mount a file system

SYNOPSIS

```
#include <sys/types.h>
#include <sys/mount.h>

int mount (spec, dir, mflag, fstyp, dataptr, datalen)
char *spec, *dir;
int mflag, fstyp;
caddr_t dataptr;
int datalen;
```

DESCRIPTION

The *mount* command requests that a removable file system contained on the block special file identified by *spec* be mounted on the directory identified by *dir*. *spec* and *dir* are pointers to path names. *fstyp* is the file system type number. The *sysfs*(2) system call can be used to determine the file system type number. If the **MS_FSS** flag bit of *mflag* is off, the file system type defaults to **root** file system type. If the bit is on, then *fstyp* is used to indicate the file system type. Additionally, if the **MS_DATA** flag is on in *mflag*, then *dataptr* and *datalen* are used to pass mount parameters to the system. If **MS_DATA** is off or if either *dataptr* or *datalen* is zero, it means that there is no additional data. In the normal case of a local *mount*, *dataptr* should be NULL. When mounting an NFS file system, *dataptr* should point to a structure that describes the NFS mount options.

Upon successful completion, references to the file *dir* refer to the **root** directory on the mounted file system.

The low-order bit of *mflag* is used to control write permission on the mounted file system. If **1**, writing is forbidden; otherwise writing is permitted according to individual file accessibility.

The *mount* command may be invoked only by the superuser. It is intended for use only by the *mount*(1M) utility.

The *mount* command will fail if one or more of the following is true:

[EPERM]	The effective user ID is not superuser.
[ENOENT]	Any of the named files does not exist.
[ENOTDIR]	A component of a path prefix is not a directory.
[EREMOTE]	The *spec* argument is remote and cannot be mounted.
[ENOLINK]	The *path* argument points to a remote machine and the link to that machine is no longer active.
[EMULTIHOP]	Components of *path* require hopping to multiple remote machines.
[ENOTBLK]	The *spec* argument is not a block special device.
[ENXIO]	The device associated with *spec* does not exist.
[ENOTDIR]	*dir* is not a directory.
[EFAULT]	The *spec* or *dir* arguments point outside the allocated address space of the process.

[EBUSY] The *dir* argument is currently mounted on, is someone's current working directory, or is otherwise busy.

[EBUSY] The device associated with *spec* is currently mounted.

[EBUSY] There are no more mount table entries.

[EROFS] The *spec* argument is write-protected and *mflag* requests write permission.

[ENOSPC] The file system state in the superblock is not FsOKAY and *mflag* requests write permission.

[EINVAL] The superblock has a bad magic number, the *fstyp* is invalid, or *mflag* is not valid.

SEE ALSO

sysfs(2), umount(2), fs(4) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual.*

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise, a value of −1 is returned, and *errno* is set to indicate the error.

NAME

intro – introduction to RPC library functions

DESCRIPTION

This section describes functions that may be found in various libraries. The library functions implement the RPC and XDR primitives. All of these functions are accessible from either the RPC library, *librpc*, the RPC services library, *librpcsvc*, or the database manager library, *libdbm*. The link editor *ld*(1) and the C compiler *cc*(1) search these libraries under the **–l***libname* option. The RPC library also includes some of the functions described in Section 2.

List of Functions

Name	*Appears on Entry*	*Description*
bindresvport	bindresvport(3N)	bind to a reserved port
dbminit	dbm(3X)	open database
fetch	dbm(3X)	retrieve datum under key
store	dbm(3X)	store datum under key
delete	dbm(3X)	delete datum and key
firstkey	dbm(3X)	find first key
nextkey	dbm(3X)	find next key
getrpcent	getrpcent(3N)	get RPC entry
getrpcbynumber	getrpcent(3N)	get RPC entry by number
getrpcbyname	getrpcent(3N)	get RPC entry by name
setrpcent	getrpcent(3N)	rewind the rpc file
endrpcent	getrpcent(3N)	close the rpc file
getrpcport	getrpcport(3N)	get RPC port number
xdr_rex_start	rex(3)	XDR a REX start message
xdr_rex_result	rex(3)	XDR a REX result message
xdr_rex_ttymode	rex(3)	XDR a REX tty modes message
xdr_rex_ttysize	rex(3)	XDR a REX tty size message
auth_destroy	rpc(3N)	destroy authentication handle
authnone_create	rpc(3N)	create authentication handle
authunix_create	rpc(3N)	create authentication handle
authunix_create_default	rpc(3N)	invoke authunix_create
callrpc	rpc(3N)	call a remote procedure
clnt_broadcast	rpc(3N)	broadcast remote procedure call
clnt_call	rpc(3N)	call a remote procedure
clnt_destroy	rpc(3N)	destroy client handle
clnt_create	rpc(3N)	generic client handle creation
clnt_control	rpc(3N)	control client handle
clnt_freeres	rpc(3N)	free data allocated by RPC/XDR
clnt_geterr	rpc(3N)	get error information
clnt_pcreateerror	rpc(3N)	print error information
clnt_perrno	rpc(3N)	print error information
clnt_perror	rpc(3N)	print error information
clnt_spcreateerror	rpc(3N)	string print error information
clnt_sperrno	rpc(3N)	string print error information
clnt_sperror	rpc(3N)	string print error information
clntraw_create	rpc(3N)	client handle creation
clnttcp_create	rpc(3N)	client handle creation
clntudp_create	rpc(3N)	client handle creation
get_myaddress	rpc(3N)	return the local IP address
pmap_getmaps	rpc(3N)	return current RPC program-to-port maps

pmap_getport	rpc(3N)	return port number for RPC service
pmap_rmtcall	rpc(3N)	indirect remote procedure call
pmap_set	rpc(3N)	establish a program-to-port mapping
pmap_unset	rpc(3N)	destroy a program-to-port mapping
registerrpc	rpc(3N)	register procedure with RPC
rwall	rwall(3N)	print string to all users on a host
svc_destroy	rpc(3N)	destroy a service handle
svc_freeargs	rpc(3N)	free data allocated by RPC/XDR
svc_getargs	rpc.3N)	decode the arguments to an RPC
svc_getcaller	rpc(3N)	get the network of the caller
svc_getreqset	rpc(3N)	get RPC request
svc_getreq	rpc(3N)	get RPC request
svc_register	rpc(3N)	register an RPC service procedure
svc_run	rpc(3N)	get RPC requests
svc_sendreply	rpc(3N)	send replies to an RPC
svc_unregister	rpc(3N)	unregister an RPC service procedure
svcerr_auth	rpc(3N)	return service error
svcerr_decode	rpc(3N)	return service error
svcerr_noproc	rpc(3N)	return service error
svcerr_noprog	rpc(3N)	return service error
svcerr_progvers	rpc(3N)	return service error
svcerr_systemerr	rpc(3N)	return service error
svcerr_weakauth	rpc(3N)	return service error
svcraw_create	rpc(3N)	create service handle
svctcp_create	rpc(3N)	create service handle
svcfd_create	rpc(3N)	create service handle
svcudp_create	rpc(3N)	create service handle
xdr_accept_reply	rpc(3N)	XDR an accepted reply
xdr_authunix_parms	rpc(3N)	XDR UNIX System credentials
xdr_callhdr	rpc(3N)	XDR the RPC call header
xdr_callmsg	rpc(3N)	XDR an RPC call message
xdr_opaque_auth	rpc(3N)	XDR opaque authentication parameters
xdr_pmap	rpc(3N)	XDR parameters to portmapper procedures
xdr_pmaplist	rpc(3N)	XDR a list of port mappings
xdr_rejected_reply	rpc(3N)	XDR a rejected reply
xdr_replymsg	rpc(3N)	XDR an RPC reply message
xprt_register	rpc(3N)	register an RPC service transport handle
xprt_unregister	rpc(3N)	unregister an RPC service transport handle
xdr_array	xdr(3N)	XDR an C array of objects
xdr_bool	xdr(3N)	XDR an boolean
xdr_bytes	xdr(3N)	XDR a counted byte string
xdr_char	xdr(3N)	XDR a C character
xdr_destroy	xdr(3N)	destroy an XDR stream
xdr_double	xdr(3N)	XDR a C double
xdr_enum	xdr(3N)	XDR a C enum
xdr_float	xdr(3N)	XDR a C float
xdr_free	xdr(3N)	generic XDR free routine
xdr_getpos	xdr(3N)	get current postion of XDR stream
xdr_inline	xdr(3N)	allocate space for inline XDR operation
xdr_int	xdr(3N)	XDR a C integer
xdr_long	xdr(3N)	XDR a C long
xdrmem_create	xdr(3N)	create an XDR stream
xdr_opaque	xdr(3N)	XDR an opaque object

xdr_pointer	xdr(3N)	XDR a C pointer
xdrrec_create	xdr(3N)	create an XDR stream
xdrrec_endofrecord	xdr(3N)	mark end of record on XDR stream
xdrrec_eof	xdr(3N)	mark end of file on XDR stream
xdrrec_skiprecord	xdr(3N)	skip rest of XDR record
xdr_reference	xdr(3N)	XDR a C pointer
xdr_setpos	xdr(3N)	set current position on XDR stream
xdr_short	xdr(3N)	XDR a C short
xdrstdio_create	xdr(3N)	create an XDR stream
xdr_string	xdr(3N)	XDR a C string
xdr_u_char	xdr(3N)	XDR a C unsigned character
xdr_u_int	xdr(3N)	XDR a C unsigned integer
xdr_u_long	xdr(3N)	XDR a C unsigned long
xdr_u_short	xdr(3N)	XDR a C unsigned short
xdr_union	xdr(3N)	XDR a discriminated union of choices
xdr_vector	xdr(3N)	XDR a C fixed length array
xdr_void	xdr(3N)	XDR nothing
xdr_wrapstring	xdr(3N)	XDR a C string

FILES

/usr/lib/librpc.a	the RPC library
/usr/lib/librpcsvc.a	the RPC services library
/usr/lib/libdbm.a	the DBM library

SEE ALSO

cc(1), ld(1), nm(1), intro(2) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual.*

NAME

bindresvport – bind a socket to a privileged IP port

SYNOPSIS

```
#include <sys/types.h>
#include <netinet/in.h>

bindresvport(sd, sin)
int sd;
struct sockaddr_in *sin;
```

DESCRIPTION

The *bindresvport* routine is used to bind a socket descriptor to a privileged IP port, that is, a port number in the range 0-1023.

DIAGNOSTICS

Upon successful completion a value of 0 is returned. Otherwise a value of −1 is returned, and *errno* is set to indicate the error.

Only **root** can bind to a privileged port; this call will fail for all other users.

NAME

dbminit, fetch, store, delete, firstkey, nextkey – database subroutines

SYNOPSIS

```
#include <rpcsvc/dbm.h>

dbminit(file)
char *file;

datum fetch(key)
datum key;

store(key, content)
datum key, content;

delete(key)
datum key;

datum firstkey()

datum nextkey(key)
datum key;

dbmclose()
```

DESCRIPTION

These functions maintain key/content pairs in a database. The functions will handle very large (a billion blocks) databases and will access a keyed item in one or two file system accesses. The functions are obtained with the loader option **–ldbm**.

*key*s and *content*s are described by the *datum* typedef. A *datum* specifies a string of *dsize* bytes pointed to by *dptr*. Arbitrary binary data, as well as normal ASCII strings, are allowed. The database is stored in two files. One file is a directory containing a bit map and has **.dir** as its suffix. The second file contains all data and has **.pag** as its suffix.

Before a database can be accessed, it must be opened by *dbminit*. At the time of this call, the files *file*.**dir** and *file*.**pag** must exist. (An empty database is created by creating zero-length **.dir** and **.pag** files.)

Once open, the data stored under a key is accessed by *fetch* and data is placed under a key by *store*. A key (and its associated contents) is deleted by *delete*. A linear pass through all keys in a database may be made, in an (apparently) random order, by use of *firstkey* and *nextkey*. *firstkey* will return the first key in the database. With any key, *nextkey* will return the next key in the database. This code will traverse the database:

```
for (key = firstkey(); key.dptr != NULL; key = nextkey(key))
```

A database may be closed by calling *dbmclose*. You must close a database before opening a new one.

DIAGNOSTICS

All functions that return an *int* indicate errors with negative values. A zero return indicates "OK." Routines that return a *datum* indicate errors with a null (0) *dptr*.

BUGS

The **.pag** file will contain holes so that its apparent size is about four times its actual content. Older UNIX Systems may create real file

blocks for these holes when touched. These files cannot be copied by normal means (*cp*, *cat*, *tp*, *tar*, or *ar*) without filling in the holes.

dptr pointers returned by these subroutines point into static storage that is changed by subsequent calls.

The sum of the sizes of a key/content pair must not exceed the internal block size (currently 1024 bytes). Moreover, all key/content pairs that hash together must fit on a single block. *store* will return an error in the event that a disk block fills with inseparable data.

delete does not physically reclaim file space, although it does make it available for reuse.

The order of keys presented by *firstkey* and *nextkey* depends on a hashing function, not on anything interesting.

There are no interlocks and no reliable cache flushing; thus concurrent updating and reading is risky.

NAME

getrpcent, getrpcbyname, getrpcbynumber – get rpc entry

SYNOPSIS

```
#include <rpc/netdb.h>

struct rpcent *getrpcent()

struct rpcent *getrpcbyname(name)
char *name;

struct rpcent *getrpcbynumber(number)
int number;

setrpcent(stayopen)
int stayopen

endrpcent()
```

DESCRIPTION

The *getrpcent*, *getrpcbyname*, and *getrpcbynumber* commands each return a pointer to an object with the following structure containing the broken-out fields of a line in the RPC program number database, **/etc/rpc**.

```
struct  rpcent {
        char    *r_name;        /* name of server for this */
                                /* rpc program */
        char    **r_aliases;    /* alias list */
        long    r_number;       /* rpc program number */
};
```

The members of this structure are:

r_name The name of the server for this RPC program.

r_aliases A zero-terminated list of alternate names for the RPC program.

r_number The RPC program number for this service.

The *getrpcent* command reads the next line of the file, opening the file if necessary.

The *setrpcent* command opens and rewinds the file. If the *stayopen* flag is nonzero, the net database will not be closed after each call to *getrpcent* (either directly, or indirectly through one of the other *getrpc* calls).

The *endrpcent* command closes the file.

The *getrpcbyname* and *getrpcbynumber* commands sequentially search from the beginning of the file until a matching RPC program name or program number is found, or until EOF is encountered.

FILES

/etc/rpc

SEE ALSO

rpcinfo(1M), rpc(4).

DIAGNOSTICS

A null pointer (0) is returned on EOF or error.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

getrpcport – get RPC port number

SYNOPSIS

int getrpcport(host, prognum, versnum, proto)
char *host;
int prognum, versnum, proto;

DESCRIPTION

The *getrpcport* routine returns the port number for version *versnum* of the RPC program *prognum* running on *host* and using protocol *proto*. It returns 0 if it cannot contact the portmapper or if *prognum* is not registered. If *prognum* is registered but not with version *versnum*, it will still return a port number (for some version of the program) indicating that the program is indeed registered. The version mismatch will be detected upon the first call to the service.

NAME

rex – remote execution protocol

SYNOPSIS

#include <rpcsvc/rex.h>

DESCRIPTION

This server executes commands remotely. The working directory and environment of the command can be specified, and the standard input and output of the command can be arbitrarily redirected. An option is provided for interactive I/O for programs that expect to be running on terminals. Note that this service is only provided with the TCP transport.

RPC Information

program number:

REXPROG

xdr routines:

```
int xdr_rex_start(xdrs, start);
        XDR *xdrs;
        struct rex_start *start;
int  xdr_rex_result(xdrs, result);
        XDR *xdrs;
        struct rex_result *result;
int xdr_rex_ttymode(xdrs, mode);
        XDR *xdrs;
        struct rex_ttymode *mode;
int xdr_rex_ttysize(xdrs, size);
        XDR *xdrs;
        struct ttysize *size;
```

procs:

REXPROC_START
: Takes *rex_start* structure, starts a command executing, and returns a *rex_result* structure.

REXPROC_WAIT
: Takes no arguments, waits for a command to finish executing, and returns a *rex_result* structure.

REXPROC_MODES
: Takes a *rex_ttymode* structure and sends the tty modes.

REXPROC_WINCH
: Takes a ttysize structure and sends window size information.

versions:

REXVERS_ORIG
: Original version

structures:

```
struct B_sgttyb {
        char    bsg_ispeed;     /* input speed     */
        char    bsg_ospeed;     /* output speed    */
        char    bsg_erase;      /* erase character */
        char    bsg_kill;       /* kill character  */
        short   bsg_flags;
};
```

```
struct tchars {
        char    t_intrc;        /* interrupt    */
        char    t_quitc;        /* quit         */
        char    t_startc;       /* start output */
        char    t_stopc;        /* stop output  */
        char    t_eofc;         /* end-of-file  */
        char    t_brkc;         /* input delimiter (like nl) */
};

struct ltchars {
        char    t_suspc;        /* stop process signal          */
        char    t_dsuspc;       /* delayed stop process signal  */
        char    t_rprntc;       /* reprint line                 */
        char    t_flushc;       /* flush output (toggles)       */
        char    t_werasc;       /* word erase                   */
        char    t_lnextc;       /* literal next character       */
};

#define REX_INTERACTIVE     1       /* Interactive mode */
struct rex_start {
        char    **rst_cmd;      /* list of command and args     */
        char    *rst_host;      /* working directory host name  */
        char    *rst_fsname;    /* working directory file system */
                                /* name  */
        char    *rst_dirwithin; /* working directory within file */
                                /* system */
        char    **rst_env;      /* list of environment          */
        ushort  rst_port0;      /* port for stdin               */
        ushort  rst_port1;      /* port for stdin               */
        ushort  rst_port2;      /* port for stdin               */
        ulong   rst_flags;      /* options - see #defines above */
};

struct rex_result {
        int     rlt_stat;       /* integer status code          */
        char    *rlt_message;   /* string message for human     */
                                /* consumption */
};

struct rex_ttymode {
        struct B_sgttyb basic;  /* Berkeley unix tty flags      */
        struct tchars more;     /* interrupt, kill characters, etc. */
        struct ltchars yetmore; /* special Berkeley characters  */
        ulong andmore;          /* and Berkeley modes           */
};

struct ttysize {
        int     ts_lines;       /* number of lines on terminal  */
        int     ts_cols;        /* number of columns on terminal */
};
```

SEE ALSO

on(1), rexd(1M).

NAME

rpc – library routines for remote procedure calls

DESCRIPTION AND SYNOPSIS

These routines allow C programs to make procedure calls on other machines across the network. First, the client calls a procedure to send a data packet to the server. Upon receipt of the packet, the server calls a dispatch routine to perform the requested service and then sends back a reply. Finally, the procedure call returns to the client.

#include <rpc/rpc.h>

void
auth_destroy(auth)
AUTH *auth;

A macro that destroys the authentication information associated with *auth*. Destruction usually involves deallocation of private data structures. The use of *auth* is undefined after calling **auth_destroy**.

AUTH *
authnone_create()

Creates and returns an RPC authentication handle that passes nonusable authentication information with each remote procedure call. This is the default authentication used by RPC.

AUTH *
authunix_create(host, uid, gid, len, aup_gids)
char *host;
int uid, gid, len, *aup_gids;

Creates and returns an RPC authentication handle that contains authentication information. The parameter *host* is the name of the machine on which the information was created; *uid* is the user ID; *gid* is the user's current group ID; *len* and **aup_gids** refer to a counted array of groups to which the user belongs. It is easy to impersonate a user.

AUTH *
authunix_create_default()

Calls **authunix_create** with the appropriate parameters.

callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
char *host;
ulong prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;

Calls the remote procedure associated with *prognum*, *versnum*, and *procnum* on the machine, *host*. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s). The *inproc* call is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. This routine returns 0 if it succeeds or the value of *enum* **clnt_stat** cast to an integer if it

fails. The routine **clnt_perrno** is handy for translating failure statuses into messages.

WARNING: Calling remote procedures with this routine uses UDP/IP as a transport; see **clntudp_create** for restrictions. You do not have control of timeouts or authentication using this routine.

enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)
ulong prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
resultproc_t eachresult;

Like *callrpc*, except the call message is broadcast to all locally connected broadcast nets. Each time it receives a response, this routine calls *eachresult*, whose form is:

eachresult(out, addr)
char *out;
struct sockaddr_in *addr;

where *out* is the same as *out* passed to **clnt_broadcast**, except that the remote procedure's output is decoded there; *addr* points to the address of the machine that sent the results. If *eachresult* returns 0, **clnt_broadcast** waits for more replies; otherwise, it returns with appropriate status.

WARNING: Broadcast sockets are limited in size to the maximum transfer unit of the data link. For Ethernet this value is 1500 bytes.

enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
CLIENT *clnt; ulong procnum;
xdrproc_t inproc, outproc;
char *in, *out;
struct timeval tout;

A macro that calls the remote procedure *procnum* associated with the client handle, *clnt*, which is obtained with an RPC client creation routine such as **clnt_create**. The parameter *in* is the address of the procedure's argument(s), and *out* is the address of where to place the result(s). The *inproc* call is used to encode the procedure's parameters, and *outproc* is used to decode the procedure's results. The *tout* call is the time allowed for results to come back.

clnt_destroy(clnt)
CLIENT *clnt;

A macro that destroys the client's RPC handle. Destruction usually involves deallocation of private data structures, including *clnt* itself. Use of *clnt* is undefined after calling **clnt_destroy**. If the RPC library opened the associated socket, it will close it also. Otherwise, the socket remains open.

CLIENT *
clnt_create (host, prog, vers, proto)
char *host;
ulong prog, vers;
char *proto;

Generic client creation routine. *host* identifies the name of the remote host where the server is located. *proto* indicates which kind of transport protocol to use. The currently supported values for this field are *udp* and *tcp*. Default timeouts are set, but they can be modified using **clnt_control**.

WARNING: Using UDP has its shortcomings. Since UDP-based RPC messages can only hold up to 8 KB of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

bool_t
clnt_control(cl, req, info)
CLIENT *cl;
char *info;

A macro used to change or retrieve various information about a client object. *req* indicates the type of operation, and *info* is a pointer to the information. For both UDP and TCP, the supported values of *req* and their argument types and what they do are:

CLSET_TIMEOUT	struct timeval	set total timeout
CLGET_TIMEOUT	struct timeval	get total timeout

Note that if you set the timeout using **clnt_control**, the timeout parameter passed to **clnt_call** will be ignored in all future calls.

CLGET_SERVER_ADDR	struct sockaddr	get server's address

The following operations are valid for UDP only:

CLSET_RETRY_TIMEOUT	struct timeval	set the retry timeout
CLGET_RETRY_TIMEOUT	struct timeval	get the retry timeout

The retry timeout is the time that UDP RPC waits for the server to reply before retransmitting the request.

clnt_freeres(clnt, outproc, out)
CLIENT *clnt;
xdrproc_t outproc;
char *out;

A macro that frees any data allocated by the RPC/XDR system when it decoded the results of an RPC call. The parameter *out* is the address of the results, and *outproc* is the XDR routine describing the results in simple primitives. This routine returns 1 if the results were successfully freed, and 0 otherwise.

void
clnt_geterr(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;

A macro that copies the error structure out of the client handle to the structure at address *errp*.

void
clnt_pcreateerror(s)
char *s;

Prints a message to standard error indicating why a client RPC handle could not be created. The message is prepended with the string *s* and a colon. This is used when a **clnt_create**, **clntraw_create**, **clnttcp_create**, or **clntudp_create** call fails.

void
clnt_perrno(stat)
enum clnt_stat stat;

Prints a message to standard error corresponding to the condition indicated by *stat*. Used after *callrpc*.

clnt_perror(clnt, s)
CLIENT *clnt;
char *s;

Prints a message to standard error indicating why an RPC call failed; *clnt* is the handle used to do the call. The message is prepended with string *s* and a colon. Used after **clnt_call**.

char *
clnt_spcreateerror
char *s;

Like **clnt_pcreateerror**, except that it returns a string instead of printing to the standard error.

Bugs: Returns a pointer to static data that is overwritten on each call.

char *
clnt_sperrno(stat)
enum clnt_stat stat;

Takes the same arguments as **clnt_perrno**, but instead of sending a message to the standard error indicating why an RPC call failed, it returns a pointer to a string that contains the message. The string ends with a new-line character.

clnt_sperrno is used instead of **clnt_perrno** if the program does not have a standard error (as a program running as a server quite likely does not), if the programmer does not want the message to be output with *printf*, or if a message format different than that supported by **clnt_perrno** is to be used. Note that unlike **clnt_sperror** and **clnt_spcreaterror**, **clnt_sperrno** does not return pointer to static data, so the result will not get overwritten on each call.

char *
clnt_sperror(rpch, s)
CLIENT *rpch;
char *s;

Like **clnt_perror**, except that (like **clnt_sperrno**) it returns a string instead of printing to standard error.

Bugs: Returns a pointer to static data that is overwritten on each call.

CLIENT *
clntraw_create(prognum, versnum)
ulong prognum, versnum;

This routine creates a toy RPC client for the remote program *prognum*, version *versnum*. Since the transport used to pass messages to the service is actually a buffer within the process's address space, the corresponding RPC server should live in the same address space (see **svcraw_create**). This allows simulation of RPC and acquisition of RPC overheads, such as round trip times, without any kernel interference. This routine returns NULL if it fails.

CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
struct sockaddr_in *addr;
ulong prognum, versnum;
int *sockp;
uint sendsz, recvsz;

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses TCP/IP as a transport. The remote program is located at Internet address ***addr*. If **addr->sin_port** is 0, then it is set to the actual port that the remote program is listening on (the remote *portmap* service is consulted for this information). The parameter *sockp* is a socket; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets *sockp*. Since TCP-based RPC uses buffered I/O, the user may specify the size of the send and receive

buffers with the parameters *sendsz* and *recvsz*; values of 0 choose suitable defaults. This routine returns NULL if it fails.

CLIENT *
clntudp_create(addr, pronum, versnum, wait, sockp)
struct sockaddr_in *addr;
ulong prognum, versnum;
struct timeval wait;
int *sockp;

This routine creates an RPC client for the remote program *prognum*, version *versnum*; the client uses the UDP/IP as a transport. The remote program is located at Internet address *addr*. If **addr->sin_port** is 0, then it is set to the actual port that the remote program is listening on (the remote *portmap* service is consulted for this information). The parameter *sockp* is a socket; if it is **RPC_ANYSOCK**, then this routine opens a new one and sets *sockp*. The UDP transport resends the call message in intervals of *wait* time until a response is received or until the call times out. The total time for the call to time out is specified by **clnt_call**.

WARNING: Since UDP-based RPC messages can only hold up to 8 KB of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

void
get_myaddress(addr)
struct sockaddr_in *addr;

Stuffs the machine's IP address into **addr*, without consulting the library routines that deal with **/etc/hosts**. The port number is always set to *htons*(PMAPPORT).

struct pmaplist *
pmap_getmaps(addr)
struct sockaddr_in *addr;

A user interface to the *portmap* service, which returns a list of the current RPC program-to-port mappings on the host located at IP address **addr*. This routine can return NULL. The command *rpcinfo* **-p** uses this routine.

ushort
pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
ulong prognum, versnum, protocol;

A user interface to the *portmap* service, which returns the port number on which waits a service that supports program number *prognum*, version *versnum*, and speaks the transport protocol associated with *protocol*. The value of *protocol* is most likely **IPPROTO_UDP** or **IPPROTO_TCP**. A return value of 0 means that the mapping does not exist or that the RPC system failed to contact the remote *portmap* service. In the latter case, the global variable **rpc_createerr** contains the RPC status.

enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum, inproc, in, outproc, out, tout, portp) struct sockaddr_in *addr;
ulong prognum, versnum, procnum;
char *in, *out;
xdrproc_t inproc, outproc;
struct timeval tout;
ulong *portp;

A user interface to the *portmap* service, which instructs *portmap* on the host at IP address ***addr* to make an RPC call on your behalf to a procedure on that host. The parameter ***portp* is modified to the program's port number if the procedure succeeds. The definitions of other parameters are discussed in *callrpc* and **clnt_call**. This procedure should be used for a "ping" and nothing else. See also **clnt_broadcast**.

pmap_set(prognum, versnum, protocol, port)
ulong prognum, versnum, protocol;
ushort port;

A user interface to the *portmap* service, which establishes a mapping between the triple [*prognum, versnum, protocol*] and *port* on the machine's *portmap* service. The value of *protocol* is most likely **IPPROTO_UDP** or **IPPROTO_TCP**. This routine returns 1 if it succeeds, 0 otherwise. This is automatically done by **svc_register**.

pmap_unset(prognum, versnum)
ulong prognum, versnum;

A user interface to the *portmap* service, which destroys all mapping between the triple [*prognum, versnum,* *] and *ports* on the machine's *portmap* service. This routine returns 1 if it succeeds, 0 otherwise.

registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
ulong prognum, versnum, procnum;
char *(*procname) () ;
xdrproc_t inproc, outproc;

Registers procedure *procname* with the RPC service package. If a request arrives for program *prognum*, version *versnum*, and procedure *procnum*, *procname* is called with a pointer to its parameter(s). The *progname* command should return a pointer to its static result(s). The *inproc* command is used to decode the parameters, while *outproc* is used to encode the results. This routine returns 0 if the registration succeeded, −1 otherwise.

WARNING: Remote procedures registered in this form are accessed using the UDP/IP transport; see **svcudp_create** for restrictions.

struct rpc_createerr
rpc_createerr;

A global variable whose value is set by any RPC client creation routine that does not succeed. Use the routine **clnt_pcreateerror** to print the reason why.

svc_destroy(xprt)
SVCXPRT * xprt;

A macro that destroys the RPC service transport handle *xprt*. Destruction usually involves deallocation of private data structures, including *xprt* itself. Use of *xprt* is undefined after calling this routine.

fd_set svc_fdset;

A global variable reflecting the RPC service side's read file descriptor bit mask; it is suitable as a parameter to the *select* system call. This is only of interest if a service implementor does not call **svc_run**, but instead implements custom asynchronous event processing. This variable is read-only (do not pass its address to *select*!), yet it may change after calls to **svc_getreqset** or any creation routines.

int svc_fds;

Similar to **svc_fedset**, but limited to 32 descriptors. This interface is made obsolete by **svc_fdset**.

svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;

A macro that frees any data allocated by the RPC/XDR system when it decoded the arguments to a service procedure using **svc_getargs**. This routine returns 1 if the results were successfully freed, 0 otherwise.

svc_getargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;

A macro that decodes the arguments of an RPC request associated with the RPC service transport handle *xprt*. The parameter *in* is the address where the arguments will be placed. The XDR routine *inproc* is used to decode the arguments. This routine returns 1 if decoding succeeds, 0 otherwise.

struct sockaddr_in
svc_getcaller(xprt)
SVCXPRT *xprt;

The approved way of getting the network address of the caller of a procedure associated with the RPC service transport handle *xprt*.

svc_getreqset(rdfds)
fd_set *rdfds;

This routine is only of interest if a service implementor does not call **svc_run**, but instead implements custom asynchronous event processing. It is called when the *select* system call has determined that an RPC request has arrived on some RPC socket(s); *rdfds* is the resultant read file descriptor bit mask. The routine returns when all sockets associated with the value of *rdfds* have been serviced.

svc_getreq(rdfds)
int rdfds;

Similar to **svc_getreqset**, but limited to 32 descriptors. This interface is made obsolete by **svc_getreqset**.

svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
ulong prognum, versnum;
void (*dispatch) ();
ulong protocol;

Associates *prognum* and *versnum* with the service dispatch procedure *dispatch*. If *protocol* is 0, the service is not registered with the *portmap* service. If *protocol* is not 0, a mapping of the triple [*prognum, versnum, protocol*] to **xprt->xp_port** is established with the local *portmap* service (generally *protocol* is 0, **IPPROTO_UDP** or **IPPROTO_TCP**). The procedure *dispatch* has the following form:

dispatch(request, xprt)
struct svc_req *request;
SVCXPRT *xprt;

The **svc_register** routine returns 1 if it succeeds, 0 otherwise.

svc_run()

This routine never returns. It waits for RPC requests to arrive and calls the appropriate service procedure using **svc_getreq** when one does. This procedure is usually waiting for a *select* system call to return.

svc_sendreply(xprt, outproc, out)
SVCXPRT *xprt;
xdrproc_t outproc;
char *out;

Called by an RPC service's dispatch routine to send the results of a remote procedure call. The parameter *xprt* is the request's associated transport handle, *outproc* is the XDR routine that is used to encode the results, and *out* is the address of the results. This routine returns 1 if it succeeds, 0 otherwise.

void
svc_unregister(prognum, versnum)
ulong prognum, versnum;

Removes all mapping of the double [*prognum, versnum*] to dispatch routines and of the triple [*prognum, versnum, **] to port number.

void
svcerr_auth(xprt, why)
SVCXPRT *xprt;
enum auth_stat why;

Called by a service dispatch routine that refuses to perform a remote procedure call due to an authentication error.

void
svcerr_decode(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine that cannot successfully decode its parameters. See also **svc_getargs**.

void
svcerr_noproc(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine that does not implement the procedure number that the caller requests.

void
svcerr_noprog(xprt)
SVCXPRT *xprt;

Called when the desired program is not registered with the RPC package. Service implementors usually do not need this routine.

void
svcerr_progvers(xprt)
SVCXPRT *xprt;

Called when the desired version of a program is not registered with the RPC package. Service implementors usually do not need this routine.

void
svcerr_systemerr(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine when it detects a system error not covered by a particular protocol. For example, if a service can no longer allocate storage, it may call this routine.

void
svcerr_weakauth(xprt)
SVCXPRT *xprt;

Called by a service dispatch routine that refuses to perform a remote procedure call due to insufficient (but correct) authentication parameters. The routine calls **svcerr_auth(xprt, AUTH_TOOWEAK)**.

SVCXPRT *
svcraw_create()

This routine creates a toy RPC service transport to which it returns a pointer. Since the transport is really a buffer within the process's address space, the corresponding RPC client should live in the same address space; see **clntraw_create**. This routine allows simulation of RPC and acquisition of RPC overheads (such as round trip times), without any kernel interference. This routine returns NULL if it fails.

SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
int sock;
uint send_buf_size, recv_buf_size;

This routine creates a TCP/IP-based RPC service transport to which it returns a pointer. The transport is associated with the socket *sock*, which may be **RPC_ANYSOCK**, in which case a new socket is created. If the socket is not bound to a local TCP port, this routine binds it to an arbitrary port. Upon completion, **xprt->xp_sock** is the transport's socket number, and **xprt->xp_port** is the transport's port number. This routine returns NULL if it fails. Since TCP-based RPC uses buffered I/O, users may specify the size of buffers; values of 0 choose suitable defaults.

void
svcfd_create(fd, sensize, recvsize)
int fd;
uint sendsize;
uint recvsize;

Creates a service on top of any open descriptor. Typically, this descriptor is a connected socket for a stream protocol such as TCP. *sendsize* and *recvsize* indicate sizes for the send and receive buffers. If they are 0, a reasonable default is chosen.

SVCXPRT *
svcudp_create(sock)
int sock;

This routine creates a UDP/IP-based RPC service transport to which it returns a pointer. The transport is associated with the socket *sock*, which may be **RPC_ANYSOCK**, in which case a new socket is created. If the socket is not bound to a local UDP port, this routine binds it to an arbitrary port. Upon completion, **xprt->xp_sock** is the transport's socket number, and **xprt->xp_port** is the transport's port number. This routine returns NULL if it fails.

WARNING: Since UDP-based RPC messages can only hold up to 8 KB of encoded data, this transport cannot be used for procedures that take large arguments or return huge results.

xdr_accepted_reply(xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;

Used for describing RPC messages externally. This routine is helpful for users who wish to generate RPC-style messages without using the RPC package.

xdr_authunix_parms(xdrs, aupp)
XDR *xdrs;
struct authunix_parms *aupp;

Used for describing UNIX System credentials externally. This routine is helpful for users who wish to generate these credentials without using the RPC authentication package.

void
xdr_callhdr(xdrs, chdr)
XDR *xdrs;
struct rpc_msg *chdr;

Used for describing RPC messages externally. This routine is helpful for users who wish to generate RPC-style messages without using the RPC package.

xdr_callmsg(xdrs, cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;

Used for describing RPC messages externally. This routine is helpful for users who wish to generate RPC-style messages without using the RPC package.

xdr_opaque_auth(xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;

Used for describing RPC messages externally. This routine is helpful for users who wish to generate RPC-style messages without using the RPC package.

xdr_pmap(xdrs, regs)
XDR *xdrs;
struct pmap *regs;

Used for describing parameters to various *portmap* procedures externally. This routine is helpful for users who wish to generate these parameters without using the *pmap* interface.

xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;

Used for describing a list of port mappings externally. This routine is helpful for users who wish to generate these parameters without using the *pmap* interface.

xdr_rejected_reply(xdrs, rr)
XDR *xdrs;
struct rejected_reply *rr;

Used for describing RPC messages externally. This routine is helpful for users who wish to generate RPC-style messages without using the RPC package.

xdr_replymsg(xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;

Used for describing RPC messages externally. This routine is helpful for users who wish to generate RPC style messages without using the RPC package.

void
xprt_register(xprt)
SVCXPRT *xprt;

After RPC service transport handles are created, they should register themselves with the RPC service package. This routine modifies the global variable **svc_fds**. Service implementors usually do not need this routine.

void
xprt_unregister(xprt)
SVCXPRT *xprt;

Before an RPC service transport handle is destroyed, it should unregister itself with the RPC service package. This routine modifies the global variable **svc_fds**. Service implementors usually do not need this routine.

SEE ALSO

xdr(3N).
"INTERACTIVE NFS Protocol Specifications and User's Guide."

NAME

rwall – write to specified remote machines

SYNOPSIS

```
#include <rpcsvc/rwall.h>

rwall(host, msg);
        char *host, *msg;
```

DESCRIPTION

host prints the string *msg* to all its users. It returns 0 if successful.

RPC Information

program number:
WALLPROG

procs:
WALLPROC_WALL
Takes string as argument (wrapstring), returns no arguments. Executes *wall* on remote host with string.

versions:
RSTATVERS_ORIG

SEE ALSO

rwall(1M), rwalld(1M).

NAME

xdr – library routines for external data representation

DESCRIPTION AND SYNOPSIS

These routines allow C programmers to describe arbitrary data structures in a machine-independent fashion. Data for remote procedure calls are transmitted using these routines.

xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
XDR *xdrs;
char **arrp;
uint *sizep, maxsize, elsize;
xdrproc_t elproc;

A filter primitive that translates between variable-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *sizep* is the address of the element count of the array. This element count cannot exceed *maxsize*. The parameter *elsize* is the *sizeof* each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form and the external representation. This routine returns 1 if it succeeds, 0 otherwise.

xdr_bool(xdrs, bp)
XDR *xdrs;
bool_t *bp;

A filter primitive that translates between booleans (C integers) and their external representations. When encoding data, this filter produces values of either 1 or 0. This routine returns 1 if it succeeds, 0 otherwise.

xdr_bytes(xdrs, sp, sizep, maxsize)
XDR *xdrs;
char **sp;
uint *sizep, maxsize;

A filter primitive that translates between counted byte strings and their external representations. The parameter *sp* is the address of the string pointer. The length of the string is located at address *sizep*; strings cannot be longer than *maxsize*. This routine returns 1 if it succeeds, 0 otherwise.

xdr_char(xdrs, cp)
XDR *xdrs;
char *cp;

A filter primitive that translates between C characters and their external representations. This routine returns 1 if it succeeds, 0 otherwise. Note that encoded characters are not packed and occupy 4 bytes each. For arrays of characters, it is worthwhile to consider **xdr_bytes**, **xdr_opaque**, or **xdr_string**.

void
xdr_destroy(xdrs)
XDR *xdrs;

A macro that invokes the *destroy* routine associated with the XDR stream *xdrs*. Destruction usually involves freeing private data structures associated with the stream. Using *xdrs* after invoking **xdr_destroy** is undefined.

xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;

A filter primitive that translates between C *double* precision numbers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;

A filter primitive that translates between C *enum*s (actually integers) and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_float(xdrs, fp)
XDR *xdrs;
float *fp;

A filter primitive that translates between C *float*s and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

void
xdr_free(proc, objp)
xdrproc_t proc;
char *objp;

Generic freeing routine. The first argument is the XDR routine for the object being freed. The second argument is a pointer to the object itself. Note that the pointer passed to this routine is *not* freed, but what it points to *is* freed (recursively).

uint
xdr_getpos(xdrs)
XDR *xdrs;

A macro that invokes the get-position routine associated with the XDR stream *xdrs*. The routine returns an unsigned integer, which indicates the position of the XDR byte stream. A desirable feature of XDR streams is that simple arithmetic works with this number, although the XDR stream instances need not guarantee this.

long *
xdr_inline(xdrs, len)
XDR *xdrs;
int len;

A macro that invokes the in-line routine associated with the XDR stream *xdrs*. The routine returns a pointer to a

contiguous piece of the stream's buffer; *len* is the byte length of the desired buffer. Note that pointer is cast to *long* *.

WARNING: *xdr_inline* may return NULL (0) if it cannot allocate a contiguous piece of a buffer. Therefore the behavior may vary among stream instances; it exists for the sake of efficiency.

xdr_int(xdrs, ip)
XDR *xdrs;
int *ip;

A filter primitive that translates between C integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;

A filter primitive that translates between C *long* integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

void
xdrmem_create(xdrs, addr, size, op)
XDR *xdrs;
char *addr;
uint size;
enum xdr_op op;

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to, or read from, a chunk of memory at location *addr* whose length is no more than *size* bytes long. The *op* determines the direction of the XDR stream (either **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**).

xdr_opaque(xdrs, cp, cnt)
XDR *xdrs;
char *cp;
uint cnt;

A filter primitive that translates between fixed-size opaque data and its external representation. The parameter *cp* is the address of the opaque object, and *cnt* is its size in bytes. This routine returns 1 if it succeeds, 0 otherwise.

xdr_pointer(xdrs, objpp, objsize, xdrobj)
XDR *xdrs;
char **objpp;
uint objsize;
xdrproc_t xdrobj;

Like **xdr_reference** in that it uses XDR's pointers, but the difference is that **xdr_pointer** serializes NULL pointers, whereas **xdr_reference** does not. Thus **xdr_pointer** can XDR recursive data structures, such as binary trees or linked lists, correctly, whereas **xdr_reference** will fail.

void
xdrrec_create(xdrs, sendsize, recvsize, handle, readit, writeit)
XDR *xdrs;
uint sendsize, recvsize;
char *handle;
int (*readit)(), (*writeit)();

This routine initializes the XDR stream object pointed to by *xdrs*. The stream's data is written to a buffer of size *sendsize*; a value of 0 indicates the system should use a suitable default. The stream's data is read from a buffer of size *recvsize*; it, too, can be set to a suitable default by passing a 0 value. When a stream's output buffer is full, *writeit* is called. Similarly, when a stream's input buffer is empty, *readit* is called. The behavior of these two routines is similar to the UNIX System calls *read* and *write*, except that *handle* is passed to the former routines as the first parameter. Note that the XDR stream's *op* field must be set by the caller.

WARNING: This XDR stream implements an intermediate record stream. Therefore, there are additional bytes in the stream to provide record boundary information.

xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;

This routine can be invoked only on streams created by **xdrrec_create**. The data in the output buffer is marked as a completed record, and the output buffer is optionally written out if *sendnow* is not 0. This routine returns 1 if it succeeds, 0 otherwise.

xdrrec_eof(xdrs)
XDR *xdrs;
int empty;

This routine can be invoked only on streams created by **xdrrec_create**. After consuming the rest of the current record in the stream, this routine returns 1 if the stream has no more input, 0 otherwise.

xdrrec_skiprecord(xdrs)
XDR *xdrs;

This routine can be invoked only on streams created by **xdrrec_create**. It tells the XDR implementation that the rest of the current record in the stream's input buffer should be discarded. This routine returns 1 if it succeeds, 0 otherwise.

xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
uint size;
xdrproc_t proc;

A primitive that provides pointer chasing within structures. The parameter *pp* is the address of the pointer, *size* is the *sizeof* the structure that **pp* points to, and *proc* is an XDR procedure that filters the structure between its C form and its external representation. This routine returns 1 if it succeeds, 0 otherwise.

WARNING: This routine does not understand NULL pointers. Use **xdr_pointer** instead.

xdr_setpos(xdrs, pos)
XDR *xdrs;
uint pos;

A macro that invokes the set position routine associated with the XDR stream *xdrs*. The parameter *pos* is a position value obtained from **xdr_getpos**. This routine returns 1 if the XDR stream could be repositioned, 0 otherwise.

WARNING: Since it is difficult to reposition some types of XDR streams, this routine may fail with one type of stream and succeed with another.

xdr_short(xdrs, sp)
XDR *xdrs;
short *sp;

A filter primitive that translates between C *short* integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

void
xdrstdio_create(xdrs, file, op)
XDR *xdrs;
FILE *file;
enum xdr_op op;

This routine initializes the XDR stream object pointed to by *xdrs*. The XDR stream data is written to, or read from, the standard I/O stream *file*. The parameter *op* determines the direction of the XDR stream (either **XDR_ENCODE**, **XDR_DECODE**, or **XDR_FREE**).

WARNING: The destroy routine associated with such XDR streams calls *fflush* on the *file* stream, but never *fclose*.

xdr_string(xdrs, sp, maxsize)
XDR *xdrs;
char **sp;
uint maxsize;

A filter primitive that translates between C strings and their corresponding external representations. Strings cannot be longer than *maxsize*. Note that *sp* is the address of the string's pointer. This routine returns 1 if it succeeds, 0 otherwise.

xdr_u_char(xdrs, ucp)
XDR *xdrs;
unsigned char *ucp;

A filter primitive that translates between *unsigned* C characters and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_u_int(xdrs, up)
XDR *xdrs;
unsigned *up;

A filter primitive that translates between C *unsigned* integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_u_long(xdrs, ulp)
XDR *xdrs;
unsigned long *ulp;

A filter primitive that translates between C *unsigned long* integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_u_short(xdrs, usp)
XDR *xdrs;
unsigned short *usp;

A filter primitive that translates between C *unsigned short* integers and their external representations. This routine returns 1 if it succeeds, 0 otherwise.

xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
int *dscmp;
char *unp;
struct xdr_discrim *choices;
bool_t (*defaultarm)(); /* may equal NULL */

A filter primitive that translates between a discriminated C *union* and its corresponding external representation. It first translates the discriminant of the union located at *dscmp*. This discriminant is always an **enum_t**. Next the union located at *unp* is translated. The parameter *choices* is a pointer to an array of **xdr_discrim** structures. Each structure contains an ordered pair of [*value, proc*]. If the union's discriminant is equal to the associated *value*, the *proc* is called to translate the union. The end of the **xdr_discrim** structure array is denoted by a routine of value NULL. If the discriminant is not found in the *choices* array, then the *defaultarm* procedure is called (if it is not NULL). Returns 1 if it succeeds, 0 otherwise.

xdr_vector(xdrs, arrp, size, elsize, elproc)
XDR *xdrs;
char *arrp;
uint size, elsize;
xdrproc_t elproc;

A filter primitive that translates between fixed-length arrays and their corresponding external representations. The parameter *arrp* is the address of the pointer to the array, while *size* is is the element count of the array. The parameter *elsize* is the *sizeof* each of the array's elements, and *elproc* is an XDR filter that translates between the array elements' C form and their external representation. This routine returns 1 if it succeeds, 0 otherwise.

xdr_void()

This routine always returns 1. It may be passed to RPC routines that require a function parameter, where nothing is to be done.

xdr_wrapstring(xdrs, sp)
XDR *xdrs;
char **sp;

A primitive that calls **xdr_string(xdrs, sp,** LASTUNSIGNED) where LASTUNSIGNED is the maximum value of an unsigned integer. **xdr_wrapstring** is handy because the RPC package passes a maximum of two XDR routines as parameters, and **xdr_string**, one of the most frequently used primitives, requires three. Returns 1 if it succeeds, 0 otherwise.

SEE ALSO

rpc(3N).
The section "XDR PROTOCOL SPECIFICATION" in the "INTERACTIVE NFS Protocol Specifications and User's Guide."
eXternal Data Representation: Sun Technical Notes.
XDR: External Data Representation Standard, RFC1014, Sun Microsystems, Inc., USC-ISI.

NAME

intro – introduction to formats of files used by Open Network Computing (ONC) commands

DESCRIPTION

This section outlines the formats of various files. The C struct declarations for the file formats are given where applicable. Usually, these structures can be found in header files under the directories **/usr/include/rpc**, **/usr/include/rpcsvc**, or **/usr/include/sys/fs/nfs**.

References of the form *name*(1M) refer to entries found in Section 1M of the "INTERACTIVE NFS Administrator's Reference Manual."

NAME

exports, xtab − directories to export to NFS clients

SYNOPSIS

/etc/exports

/etc/xtab

DESCRIPTION

The **/etc/exports** file contains entries for directories that can be exported to NFS clients. This file is read automatically by the *exportfs*(1M) command. If you change this file, you must run *exportfs* for the changes to affect the mount daemon's operation.

Only when this file is present at boot time does the NFS startup script execute *exportfs* and start the NFS file system daemon, *nfsd*(1M), and the mount daemon, *mountd*(1M).

The **/etc/xtab** file contains entries for directories that are *currently* exported. This file should only be accessed by programs using *getexportent*. (Use the **−u** option of *exportfs* to remove entries from this file.)

An entry for a directory consists of a line of the following form:

directory [−*option*[,*option*]]...

where:

directory is the path name of a directory (or file).

option is one of

ro Export the directory read-only. If not specified, the directory is exported read-write.

rw=*hostnames* [:*hostname*]...
Export the directory read-mostly. Read-mostly means read-only to most machines, but read-write to those specified. If not specified, the directory is exported read-write to all.

anon=*uid*
If a request comes from an unknown user, use *uid* as the effective user ID. Note: **root** users (uid 0) are always considered "unknown" by the NFS server, unless they are included in the "root" option below. The default value for this option is −2. Setting **anon** to −1 disables anonymous access. Note: By default NFS will accept insecure requests as anonymous; users who want extra security can disable this feature by setting **anon** to −1.

root=*hostnames* [:*hostname*]...
Give **root** access only to the **root** users from a specified *hostname*. The default is for no hosts to be granted **root** access.

access=*client* [:*client*]...
Give mount access to each *client* listed. A *client* can either be a host name or a netgroup (see *netgroup*(4)). Each *client* in the list is first checked for in the netgroup database, and then the hosts database. The default value allows any machine to mount the given directory.

A "**#**" (pound-sign) anywhere in the file indicates a comment that extends to the end of the line.

EXAMPLES

```
/usr        -access=clients              # export to my clients
/usr/local                               # export to the world
/usr2       -access=hermes:zip:tutorial  # export to only these machines
/usr/sun    -root=hermes:zip             # give root access only to these
/usr/new    -anon=0                      # give all machines root access
/usr/bin    -ro                          # export read-only to everyone
/usr/stuff  -access=zip,anon=-3,ro       # several options on one line
```

FILES

/etc/exports static export information
/etc/xtab current state of exported directories

SEE ALSO

exportfs(1M), mountd(1M), netgroup(4).

WARNINGS

You cannot export either a parent directory or a subdirectory of an exported directory that is *within the same file system*. It would be illegal, for instance, to export both **/usr** and **/usr/local** if both directories resided on the same disk partition.

NAME

netgroup – list of network groups

DESCRIPTION

netgroup defines network-wide groups, used for permission checking when doing remote mounts, remote logins, and remote shells. For remote mounts, the information in *netgroup* is used to classify machines; for remote logins and remote shells, it is used to classify users. Each line of the *netgroup* file defines a group and has the format:

groupname member1 member2 ....

where member*n* is either another group name or a triple:

(hostname, username, domainname)

Any of three fields can be empty, in which case it signifies a "wild card." Thus:

universal (,,)

defines a group to which everyone belongs. Field names that begin with something other than a letter, digit, or underscore (such as "–") work in exactly the opposite fashion. For example, consider the following entries:

justmachines	(analytic,–,sun)
justpeople	(–,babbage,sun)

The machine, *analytic*, belongs to the group, *justmachines*, in the domain, *sun*, but no users belong to it. Similarly, the user, *babbage*, belongs to the group, *justpeople*, in the domain, *sun*, but no machines belong to it.

Network groups are contained in the network information service and are accessed through these files:

/etc/yp/*domainname*/netgroup.dir
/etc/yp/*domainname*/netgroup.pag
/etc/yp/*domainname*/netgrp.usr.dir
/etc/yp/*domainname*/netgrp.usr.pag
/etc/yp/*domainname*/netgrp.hst.dir
/etc/yp/*domainname*/netgrp.hst.pag

These files can be created from **/etc/netgroup** using *makedbm*(1M).

FILES

/etc/netgroup
/etc/yp/domainname/netgroup.dir
/etc/yp/domainname/netgroup.pag
/etc/yp/domainname/netgrp.usr.dir
/etc/yp/domainname/netgrp.usr.pag
/etc/yp/domainname/netgrp.hst.dir
/etc/yp/domainname/netgrp.hst.pag

BUGS

This file is dependent on the network transport mechanism used.

SEE ALSO

makedbm(1M), ypserv(1M), ypmapxlate(4) in the *INTERACTIVE Network Information Service Guide*.

NAME

nfsd – NFS special file

SYNOPSIS

#include <sys/fs/nfs/nfs_ioctl.h>

/dev/nfsd

DESCRIPTION

The *nfsd* daemon currently handles the following *ioctl* services for the NFS client and server processes.

NIOCNFSD Starts an NFS daemon listening on the transport endpoint. This call returns only if the process is killed.

NIOCGETFH
Returns a file handle for an open file. It is only used by the NFS mount daemon and should not be used by users.

NIOCASYNCD
Implements the NFS daemon that handles asynchronous I/O for an NFS client. This call returns only if the process is killed.

NIOCSETDOMNAM
Sets the NIS domain name of the host machine.

NIOCGETDOMNAM
Returns the name of the domain for the current processor, as previously set by NIOCSETDOMNAM.

NIOCCLNTHAND
Is used to create client handles for kernel RPC clients to use. Currently, there are two kernel RPC clients: NFS and the lock manager.

NIOCEXPORTFS
Is used to add and delete *export* entries for file systems. *exportfs* exports the directory tree described by *fname*. If *uex* is null, the directory tree described by *fname* is unexported. It is only used by the NFS *exportfs* command and should not be utilized by users.

SEE ALSO

lckclnt(1M), mount(1M), nfsd(1M).

NAME

rmtab – remotely mounted file system table

DESCRIPTION

The *rmtab* file resides in directory **/etc** and contains a record of all clients that have done remote mounts of file systems from this machine. Whenever a remote *mount* is done, an entry is made in the *rmtab* file of the machine serving up that file system. The *umount* command removes entries of a remotely mounted file system. The table is a series of lines of the form:

hostname:directory

This table is used only to preserve information between crashes and is read only by *mountd*(1M) when it starts up. The *mountd* command keeps an in-core table, which it uses to handle requests from programs like *showmount*(1M) and *shutdown*(1M).

FILES

/etc/rmtab

SEE ALSO

mount(1M), mountd(1M), showmount(1M).
shutdown(1M) in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual.*
umount(2) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual.*

BUGS

Although the *rmtab* table is informative, it is not always 100 percent accurate.

NAME

rpc – rpc program number database

SYNOPSIS

/etc/rpc

DESCRIPTION

The *rpc* file contains user readable names that can be used in place of RPC program numbers. Each line has the following information:

name of server for the RPC program
RPC program number
aliases

Items are separated by any number of blanks and/or tab characters. A "#" indicates the beginning of a comment; characters up to the end of the line are not interpreted by routines that search the file.

Here is an example of an **/etc/rpc** file.

```
#
# rpc file
#
portmapper      100000      portmap sunrpc
rstat_svc       100001      rstatd rstat rup perfmeter
rusersd         100002      rusers
nfs             100003      nfsprog
ypserv          100004      ypprog
mountd          100005      mount showmount
ypbind          100007
walld           100008      rwall shutdown
yppasswdd       100009      yppasswd
etherstatd      100010      etherstat
rquotad         100011      rquotaprog quota rquota
sprayd          100012      spray
3270_mapper     100013
rje_mapper      100014
selection_svc   100015      selnsvc
database_svc    100016
rexd            100017      rex
alis            100018
sched           100019
llockmgr        100020
nlockmgr        100021
x25.inr         100022
statmon         100023
status          100024
bootparam       100026
ypupdated       100028      ypupdate
keyserv         100029      keyserver
```

FILES

/etc/rpc

SEE ALSO

getrpcent(3N).

NAME

sm, record, recover, state – statd directory and file structures

SYNOPSIS

/etc/sm/record
/etc/sm/recover
/etc/sm/state

DESCRIPTION

The **/etc/sm/record** and **/etc/sm/recover** plain text files are generated by the *statd* daemon. Each host name in **/etc/sm/record** represents the name of the machine to be monitored by the *statd* daemon. Each host name in **/etc/sm/recover** represents the name of the machine to be notified by the *statd* daemon upon its recovery.

The **/etc/sm/state** plain text file is generated by the *statd* daemon and records its current version number. This version number is incremented each time a crash or recovery takes place.

SEE ALSO

lockd(1M), statd(1M).

INTERACTIVE NFS
Administrator's Reference Manual

CONTENTS

NAME

intro – introduction to Open Network Computing (ONC) maintenance and operation commands

DESCRIPTION

This section contains information related to ONC operation and maintenance. It describes the commands used to start and stop NFS, to mount and unmount NFS file systems, and to "ping" RPC-based services (see *ping*(1M)). The various NFS/ONC daemons and utilities are also detailed. The commands used to manipulate the Network Information Service (NIS) are available in the optional Network Information Service subset.

SEE ALSO

"Introduction to the INTERACTIVE NFS Extension."
ping(1M) in the *INTERACTIVE TCP/IP Guide*.

NAME

automount − automatically mount NFS file systems

SYNOPSIS

automount [**−mnTv**] [**−D** envar=value] [**−f** master-file]
[**−M** mount-directory] [**−tl** duration] [**−tm** interval]
[**−tw** interval] [directory map [−mount-options]] ...

DESCRIPTION

automount is a daemon that automatically and transparently mounts an NFS file system as needed. It monitors attempts to access directories that are associated with an *automount* map, along with any directories or files that reside under them. When a file is to be accessed, the daemon mounts the appropriate NFS file system. You can assign a map to a directory using an entry in a direct *automount* map or by specifying an indirect map on the command line.

The *automount* daemon appears to be an NFS server to the kernel. *automount* uses the map to locate an appropriate NFS file server, exported file system, and mount options. It then mounts the file system in a temporary location and replaces the file system entry for the directory or subdirectory with a symbolic link to the temporary location. If the file system is not accessed within an appropriate interval (5 minutes by default), the daemon unmounts the file system and removes the symbolic link. If the indicated directory has not already been created, the daemon creates it, and then removes it upon exiting.

Since the name-to-location binding is dynamic, updates to an *automount* map are transparent to the user. This obviates the need to "pre-mount" shared file systems for applications that have "hard coded" references to files.

If you specify the dummy directory "/−," *automount* treats the *map* argument that follows as the name of a direct map. In a direct map, each entry associates the full path name of a mount point with a remote file system to mount.

If the *directory* argument is a path name, the *map* argument points to an indirect map. An indirect map contains a list of the subdirectories contained within the indicated *directory*. With an indirect map, it is these subdirectories that are mounted automatically.

A map can be a file or a Network Information Service (NIS) map; if a file, the *map* argument must be a full path name.

The *−mount-options* argument, when supplied, is a comma-separated list of NFS *mount*(1M) options, preceded by a "−." If mount options are specified in the indicated map, however, those in the map take precedence.

The following options are available:

−m Suppress initialization of *directory-map* pairs listed in the **auto.master** NIS database.

−n Disable dynamic mounts. With this option, references through the *automount* daemon only succeed when the target file system has been previously mounted. This can be used to prevent NFS servers from cross-mounting each other.

−T Trace. Expand each NFS call and display it on the standard output.

−v Verbose. Log status messages to the console.

−D *envar=value*
Assign *value* to the indicated *automount* (environment) variable.

−f *master-file*
Read a local file for initialization before the **auto.master** NIS map.

−M *mount-directory*
Mount temporary file systems in the named directory, instead of **/tmp_mnt**.

−tl *duration*
Specify a *duration*, in seconds, that a file system is to remain mounted when not in use. The default is 5 minutes (300 seconds).

−tm *interval*
Specify an *interval*, in seconds, between attempts to mount a file system. The default is 30 seconds.

−tw *interval*
Specify an *interval*, in seconds, between attempts to unmount file systems that have exceeded their cached times. The default is 1 minute (60 seconds).

ENVIRONMENT

Environment variables can be used within an *automount* map. For example, if **$HOME** appeared within a map, *automount* would expand it to its current value for the HOME variable.

If a reference needs to be protected from affixed characters, you can surround the variable name with curly braces.

USAGE

Map Entry Format

A simple map entry (mapping) takes the form:

directory [*−mount-options*] *location* ...

where *directory* is the full path name of the directory to mount when used in a direct map, or the basename of a subdirectory in an indirect map. *mount-options* is a comma-separated list of NFS *mount* options, and *location* specifies a remote file system from which the directory may be mounted. In the simple case, *location* takes the form:

host:pathname

Multiple *location* fields can be specified, in which case *automount* sends multiple *mount* requests; *automount* mounts the file system from the first host that replies to the *mount* request. This request is first made to the local net or subnet. If there is no response, any connected server may respond.

If *location* is specified in the form:

host:*path*:*subdir*

host is the name of the host from which to mount the file system, *path* is the path name of the directory to mount, and *subdir*, when supplied, is the name of a subdirectory to which the symbolic link is made. This can be used to prevent duplicate mounts when multiple directories in the same remote file system may be accessed. With a map for **/home** such as:

```
able    homeboy:/home/homeboy:able
baker   homeboy:/home/homeboy:baker
```

and a user attempting to access a file in **/home/able**, *automount* mounts **homeboy:/home/homeboy**, but creates a symbolic link called **/home/able** to the **able** subdirectory in the temporarily-mounted file system. If a user immediately tries to access a file in **/home/baker**, *automount* needs only to create a symbolic link that points to the **baker** subdirectory; **/home/homeboy** is already mounted. With the following map:

```
able    homeboy:/home/homeboy/able
baker   homeboy:/home/homeboy/baker
```

automount would have to mount the file system twice.

A mapping can be continued across input lines by escaping the new-line with a backslash (\). Comments begin with a **#** and end at the subsequent new-line.

Directory Pattern Matching

The "**&**" character is expanded to the value of the *directory* field for the entry in which it occurs. In this case:

```
able    homeboy:/home/homeboy:&
```

the **&** expands to **able**.

The "*" character, when supplied as the *directory* field, is recognized as the catch-all entry. Such an entry resolves to any entry not previously matched. For example, if the following entry appeared in the indirect map for **/home**:

```
*       &:/home/&
```

this would allow automatic mounts in **/home** of any remote file system whose location could be specified as:

```
hostname :/home/ hostname
```

Hierarchical Mappings

A hierarchical mapping takes the form:

directory [/[*subdirectory* [−*mount-options*] *location* ...] ...

The initial / within the "/[*subdirectory*]" is required; the optional *subdirectory* is taken as a file name relative to the *directory*. If *subdirectory* is omitted in the first occurrence, the / refers to the directory itself.

Given the direct map entry:

```
/usr/local  \
        /       -ro,intr    loco:/usr/local         alt:/usr/local  \
        /bin    -ro,intr    alt:/usr/local/bin      loco:/usr/local/bin  \
        /man    -ro,intr    loco:/usr/local/man     alt:/usr/local/man
```

automount would automatically mount **/usr/local**, **/usr/local/bin**, and **/usr/local/man**, as needed, from either **loco** or **alt**, whichever host responded first.

Direct Maps

A direct map contains mappings for any number of directories. Each directory listed in the map is automatically mounted as needed. The direct map as a whole is not associated with any single directory.

Indirect Maps

An indirect map allows you to specify mappings for the subdirectories you want to mount under the *directory* indicated on the command line. It also obscures local subdirectories for which no mapping is specified. In an indirect map, each *directory* field consists of the basename of a subdirectory to be mounted as needed.

Included Maps

The contents of another map can be included within a map with an entry of the form:

+*mapname*

mapname can either be a file name, or the name of an NIS map, or one of the special maps described below.

Special Maps

There are three special maps currently available: **−hosts**, **−passwd**, and **−null**. The **−hosts** map uses the hostname resolution facilities available on the system to locate a remote host when the hostname is specified. This map specifies mounts of all exported file systems from any host. For instance, if the following *automount* command is already in effect:

```
automount /net −hosts
```

then a reference to **/net/hermes/usr** would initiate an automatic mount of all file systems from **hermes** that *automount* can mount; references to a directory under **/net/hermes** will refer to the corresponding directory on **hermes**. The **−passwd** map uses the *passwd*(4) database to attempt to locate the home directory of a user. For example, if the following *automount* command is already in effect:

```
automount /homes −passwd
```

then if the home directory for a user has the form */dir/server/username*, and *server* matches the host system on which that directory resides, *automount* will mount the user's home directory as: **/homes**/*username*.

For this map, the tilde character "~" is recognized as a synonym for the username.

The **−null** map, when indicated on the command line, cancels a previous map for the directory indicated. It can be used to cancel a map given in **auto.master**.

Configuration and the auto.master Map

automount normally consults the **auto.master** NIS configuration map for a list of initial *automount* maps and sets up automatic mounts for them in addition to those given on the command line. If there are duplications, the command-line arguments take precedence. This configuration database contains arguments to the *automount* command, rather than mappings; unless the **−f** option is in effect, *automount* does *not* look for an **auto.master** file on the local host.

Maps given on the command line or in a local **auto.master** file specified with the **−f** option override those in the NIS **auto.master** map. For example, given the command:

automount −f /etc/auto.master /home −null /− /etc/auto.direct

and a file named **/etc/auto.master** that contains:

/homes −passwd

automount would mount home directories under **/homes** instead of **/home**, as well as mount the various directories specified in the **/etc/auto.direct** file.

FILES

/tmp_mnt	directory under which file systems are dynamically mounted
/etc/auto.master	list of master NIS configuration maps
/etc/auto.direct	list of direct automounter maps
/etc/auto.indirect	list of indirect automounter maps

SEE ALSO

mount(1M).
df(1M) in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual.*
passwd(4) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual.*

NOTES

When it receives signal number 1, **SIGHUP**, *automount* rereads the **/etc/mnttab** file to update its internal record of currently mounted file systems. If a file system mounted with *automount* is unmounted by a *umount* command (see *mount*(1M)), *automount* should be forced to reread the file.

BUGS

Shell file name expansion does not apply to objects not currently mounted.

Because *automount* is single-threaded, any request that is delayed by a slow or non-responding NFS server will delay all subsequent automatic mount requests until it completes.

Programs that read **/etc/mnttab** and then touch files that reside under automatic mount points will introduce further entries to the file.

NAME

exportfs – export and unexport directories to NFS clients

SYNOPSIS

/etc/exportfs [–avu] [–o options **] [directory]**

DESCRIPTION

exportfs makes a local directory (or file name) available for mounting over the network by NFS clients. It is normally invoked at boot time by the NFS startup script **/etc/rc3.d/S72nfs**, and uses information contained in the **/etc/exports** file to export directories (which must be specified as full path names). The superuser can run *exportfs* at any time to alter the list or characteristics of exported directories. Directories that are currently exported are listed in the file **/etc/xtab**.

With no options or arguments, *exportfs* prints out the list of directories currently exported.

The following options are available:

–a All. Export all directories listed in **/etc/exports**, or if **–u** is also specified, unexport all of the currently exported directories.

–v Verbose. Print each directory as it is exported or unexported.

–u Unexport the indicated directories.

–i Ignore the options in **/etc/exports**. Normally, *exportfs* will consult **/etc/exports** for the options associated with the exported directory.

–o *options*

Specify a comma-separated list of optional characteristics for the directory being exported. *options* can be selected from among:

ro Export the directory read-only. If not specified, the directory is exported read-write.

rw=*hostname*[**:***hostname*]...

Export the directory read-mostly. Read-mostly means exported read-only to most machines, but read-write to those specified. If not specified, the directory is exported read-write to all.

anon=*uid*

If a request comes from an unknown user, use *uid* as the effective user ID. Note: **root** users (uid 0) are always considered "unknown" by the NFS server, unless they are included in the "root" option below. The default value for this option is –2. Setting the value of **anon** to –1 disables anonymous access. Note that by default NFS accepts insecure requests as anonymous, but users who want extra security can disable this feature by setting **anon** to -1.

root=*hostname*[**:***hostname*]...

Give **root** access only to the **root** users from a specified *hostname*. The default is for no hosts to be granted **root** access.

access=*client*[:*client*]...
Give mount access to each *client* listed. A *client* can either be a host name or a netgroup (see *netgroup*(4)). Each *client* in the list is first checked for in the **/etc/netgroup** database and then in the **/etc/hosts** database. The default value allows any machine to mount the given directory.

FILES

/etc/exports	static export information
/etc/xtab	current state of exported directories
/etc/netgroup	list of network wide groups
/etc/rc3.d/S72nfs	NFS startup script

SEE ALSO
exports(4), netgroup(4).

WARNING
You cannot export a directory that is either a parent or a subdirectory of one that is currently exported and *within the same file system*. It would be illegal, for example, to export both **/usr** and **/usr/local** if both directories resided in the same disk partition.

NAME

fsirand – install random inode generation numbers

SYNOPSIS

fsirand [**–p**] special

DESCRIPTION

The *fsirand* command installs random inode generation numbers on all the inodes on device *special*. This helps increase the security of file systems exported by NFS.

The *fsirand* command must be used only on an unmounted file system that has been checked with *fsck*(1M). The only exception is that it can be used on the **root** file system in single-user mode if the system is immediately rebooted.

The following option is available:

–p Print out the generation numbers for all the inodes, but do not change the generation numbers.

SEE ALSO

fsck(1M) in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual*.

NAME

lckclnt – create lock manager client handles

SYNOPSIS

lckclnt [nclienthandles]

DESCRIPTION

The *lckclnt* command allocates connectionless transport endpoints that are used to create client handles. The *lock manager* client programs obtain a client handle for the duration of an RPC operation.

The *nclienthandles* option is the number of client handles allocated. This number limits the number of *lock manager* client operations that can be run concurrently and should be based on the load expected on the client. If additional client handles are required, more *lckclnt* processes may be started. The number of client handles available to *lock manager* client programs is the sum of the number of client handles allocated by each *lckclnt* program. Killing an *lckclnt* process reduces the available number of client handles by the amount that was initially allocated by that process.

FILES

/dev/udp UDP device node

SEE ALSO

nfsd(4).

NAME

lockd – network lock daemon

SYNOPSIS

/etc/lockd [**–t** timeout] [**–d** debuglevel]] **–g** graceperiod] [**–h** hashsize] [**–l** k2utimeout]

DESCRIPTION

The *lockd* daemon processes lock requests that are either sent locally by the kernel or remotely by another lock daemon. The *lockd* daemon forwards lock requests for remote data to the server site's lock daemon through the RPC/XDR(3N) package. The *lockd* daemon then requests the status monitor daemon, *statd*(1M), for monitor service. The reply to the lock request will not be sent to the kernel until the status daemon and the server site's lock daemon have replied.

If either the status monitor or server site's lock daemon is unavailable, the reply to a lock request for remote data is delayed until all daemons become available.

When a server recovers, it waits for a grace period for all client site *lockd*s to submit reclaim requests. Client site *lockd*s, on the other hand, are notified by the *statd* of the server recovery and promptly resubmit previously granted lock requests. If a *lockd* fails to secure a previously granted lock at the server site, the *lockd* sends SIGUSR2 to a process.

The following options are available:

–t *timeout*
: The *lockd* daemon uses *timeout* (seconds) as the interval instead of the default value (15 seconds) to retransmit lock request to the remote server.

–d *debuglevel*
: The *lockd* daemon has extensive internal reporting capabilities. A level of 2 reports significant events. A level of 4 reports internal state and all lock requests traffic; verbose.

–g *graceperiod*
: The *lockd* daemon uses *graceperiod* (seconds) as the grace period duration instead of the default value (45 seconds).

–h *hashsize*
: The *lockd* daemon uses *hashsize* hash buckets internally instead of the default of 29.

–l *k2utimeout*
: The *lockd* daemon uses *k2utimeout* (seconds) as the interval instead of the default value (2 seconds) to retransmit kernel lock manager requests. This is the timeout value used for local lock requests.

SEE ALSO

statd(1M).
fcntl(2), signal(2), lockf(3C) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual.*

NAME

mount, umount − mount and unmount file systems and remote resources

SYNOPSIS

/etc/mount [**−r**] [**−f** fstyp] special directory
/etc/mount [**−r**] [**−c**] **−d** resource directory
/etc/mount
/etc/umount special
/etc/umount directory
/etc/umount −d resource

DESCRIPTION

File systems other than **root** (/) are considered *removable* in the sense that they can be either available or unavailable to users. The *mount* command announces to the system that *special*, a block special device, or *resource,* a remote resource, is available to users from the mount point *directory.* The *directory* must exist already; it becomes the name of the **root** of the newly mounted *special* or *resource.* A unique resource may be mounted only once (no multiple mounts).

The *mount* command, when entered with arguments, adds an entry to the table of mounted devices, **/etc/mnttab.** The *umount* command removes the entry. If invoked with no arguments, *mount* prints the entire mount table. If invoked with any of the following partial argument list, *mount* searches **/etc/fstab** to fill in the missing arguments: *special,* **−d** *resource*, *directory*, or **−d** *directory*.

The following options are available:

−r Indicates that *special* or *resource* is to be mounted read-only. If *special* or *resource* is write-protected or read-only advertised, this flag must be used.

−d Indicates that *resource* is a remote resource that is to be mounted or unmounted on *directory*. To mount a remote resource, Remote File Sharing must be up and running and the resource must be advertised by a remote computer.

−c Indicates that remote reads and writes should not be cached in the local buffer pool. The **−c** option is used in conjunction with **−d.**

−f*fstyp* Indicates that *fstype* is the file system type to be mounted. If this argument is omitted, it defaults to the **root** *fstyp*. If *fstyp* is NFS, then NFS options may be added after the *fstyp* separated by commas. The available NFS options are:

soft Return error if the server does not respond.

nosuid Ignore setuid and setgid bits during *exec.*

bg Background this mount; this is recommended for automatic mounts done during system startup.

rsize=*n* Set the read buffer size to *n* bytes.

wsize=*n*
Set the write buffer size to *n* bytes.

timeo=*n*
Set the initial NFS timeout to *n* tenths of a second.

retrans=*n*
Set the number of NFS retransmissions to *n*.

port=*n* Set the server IP port number to *n*.

noac Don't cache attributes. This is necessary when close synchronization with the server is required. NOTE: Use of this option will drastically cut performance on the file system being mounted.

special Indicates the block special device that is to be mounted on *directory*. If *fstyp* is NFS, then *special* should be of the form **hostname:/pathname.**

resource Indicates the remote resource name that is to be mounted on a *directory*.

directory Indicates the directory mount point for *special* or *resource.* (The directory must already exist.)

The *umount* command announces to the system that the file system previously mounted *special* or *resource* is to be made unavailable. If invoked with *special* or **−d** *directory*, *umount* will search **/etc/fstab** to fill in the missing argument(s).

The *mount* command can be used by any user to list mounted file systems and resources. Only a superuser can mount and umount file systems.

FILES

/etc/mnttab mount table
/etc/fstab file system table

SEE ALSO

mountd(1M), nfsd(1M), showmount(1M), mount(2).
fuser(1M), setmnt(1M) in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual.*
umount(2), fstab(4), mnttab(4) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual.*

DIAGNOSTICS

If the *mount*(2) system call fails, *mount* prints an appropriate diagnostic. The *mount* command issues a warning if the file system to be mounted is currently labeled under another name. A remote resource mount will fail if the resource is not available, if Remote File Sharing is not running, or if it is advertised read-only and not mounted with **−r**.

The *umount* command fails if *special* or *resource* is not mounted or if it is busy. The *special* or *resource* command is busy if it contains an open file or some user's working directory. In such a case, you can use *fuser*(1M) to list and kill processes that are using *special* or *resource.*

WARNING

Physically removing a mounted file system diskette from the diskette drive before issuing the *umount* command damages the file system.

NAME

mountd – NFS mount request server

SYNOPSIS

/etc/mountd

DESCRIPTION

The *mountd* daemon is an RPC server that responds to file system mount requests. It reads the file **/etc/xtab**, described in *exports*(4), to determine which file systems are available to which machines and users. It also provides information as to which clients have file systems mounted. This information can be printed using the *showmount*(1M) command.

SEE ALSO

exportfs(1M), showmount(1M), exports(4), nfsd(4).
services(5) in the *INTERACTIVE TCP/IP Guide*.

NAME

/etc/init.d/nfs – NFS start/stop script

SYNOPSIS

/etc/init.d/nfs start
/etc/init.d/nfs stop

DESCRIPTION

/etc/init.d/nfs is used to start or stop the NFS software. NFS starts automatically at system startup if **/etc/init.d/nfs** is linked to **/etc/rc3.d/S***name* (*name* is installed as *72nfs* by default). Similarly, NFS stops automatically at system shutdown if **/etc/init.d/nfs** is linked to **/etc/rc0.d/K***name* (*name* is installed as *66nfs* by default). See *rc0*(1M) and *rc2*(1M) for additional information. NFS will also be shut down when leaving networking mode (run level 3) if **/etc/init.d/nfs** is linked to **/etc/rc2.d/K***name* (where *name* is *51nfs* by default).

/etc/init.d/nfs may be customized for a particular installation before it is used. The following items may need to be edited:

PATH — The supplied path may require modification if commands run by **/etc/init.d/nfs** are in other directories.

PROC 1 and PROC 2 — The PROC 1 and PROC 2 variables contain space-separated lists of names of processes to kill when executing the *stop* function. If additional daemons are used, their names can be added to this list.

Daemons — The standard NFS daemons are started at this point. Any additional daemons or other commands may be included in this section. Any of the standard daemons that are not desired may be removed or commented out.

Networking services used as a transport for NFS must be initialized before NFS is started.

FILES

/etc/rc3.d/S72nfs
/etc/rc2.d/K51nfs
/etc/rc0.d/K66nfs

SEE ALSO

lckclnt(1M), lockd(1M), mountd(1M), nfsclnt(1M), nfsd(1M), nmountall(1M), pcnfsd(1M), portmap(1M), rexd(1M), statd(1M), exports(4).
rc0(1M), rc2(1M), sh(1) in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual.*

NAME

nfsclnt – create NFS client handles

SYNOPSIS

nfsclnt [nclienthandles]

DESCRIPTION

The *nfsclnt* command allocates connectionless transport endpoints which are used to create client handles. NFS client programs obtain a client handle for the duration of an RPC operation.

The *nclienthandles* command is the number of client handles allocated. This number limits the number of NFS client operations that can be run concurrently, and should be based on the load expected on the client. If additional client handles are required, more *nfsclnt* processes may be started. The number of client handles available to NFS client programs is the sum of the number of client handles allocated by each *nfsclnt* program. Killing an *nfsclnt* process will reduce the available number of client handles by the amount that was initially allocated by that process.

FILES

/dev/udp UDP device node

SEE ALSO

nfsd(4).

NAME

nfsd, biod – NFS daemons

SYNOPSIS

/etc/nfsd [nservers]

/etc/biod [nservers]

DESCRIPTION

The *nfsd* daemon starts the NFS server daemons that handle client file system requests. *nservers* is the number of file system request daemons to start. This number should be based on the load expected on this server.

The *biod,* run on an NFS client, starts *nservers* asynchronous block I/O daemons, which do read-ahead and write-behind of blocks from the client's buffer cache.

SEE ALSO

mountd(1M), exports(4), nfsd(4).

NAME

nfsstat – Network File System statistics

SYNOPSIS

nfsstat [–csnrz]

DESCRIPTION

The *nfsstat* command displays statistical information about the Network File System (NFS) and Remote Procedure Call (RPC) interfaces to the kernel. It can also be used to reinitialize this information. If no options are given, the default is:

nfsstat –csnr

That is, print everything and reinitialize nothing.

The following options are available:

–c Display client information. Only the client side NFS and RPC information will be printed. Can be combined with the **–n** and **–r** options to print client NFS or client RPC information only.

–s Display server information. Works like the **–c** option above.

–n Display NFS information. NFS information for both the client and server side will be printed. Can be combined with the **–c** and **–s** options to print client or server NFS information only.

–r Display RPC information. Works like the **–n** option above.

–z Zero (reinitialize) statistics. Can be combined with any of the above options to zero particular sets of statistics after printing them. The user must have write permission on **/dev/kmem** for this option to work.

FILES

/unix	system namelist
/dev/kmem	kernel memory

NAME

nmountall, numountall – mount, unmount multiple file systems

SYNOPSIS

/etc/nmountall
/etc/numountall

DESCRIPTION

The *nmountall* command is used to mount NFS file systems according to entries in **/etc/fstab**. It is strongly recommended that the NFS mount option, *bg*, be used for file systems that are automatically mounted during startup. This prevents startup processing from hanging while trying to mount a file system from a very slow or dead server.

The *numountall* command causes all NFS mounted file systems to be unmounted. Processes that hold open files or have current directories on these file systems are killed by being sent a series of signals. The first signal sent is SIGHUP. One second later, SIGTERM is sent. Finally, one second later, SIGKILL is sent.

These commands may be executed only by the superuser.

FILES

File system table format:

column 1	remote file system name to be mounted
column 2	mount point directory
column 3	–r if to be mounted read-only
column 4	file system type string
column 5+	ignored

White space separates columns. Lines beginning with "#" are comments. Empty lines are ignored.

A typical file system table entry might read:

```
srcmachine:/usr/src    /usr/src    –r    NFS,soft,bg
```

SEE ALSO

mount(1M).
fuser(1M) in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual.*
umount(2), signal(2), fstab(4) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual.*

DIAGNOSTICS

The *nmountall* command prints the mount commands that it will run before it runs them.

The *numountall* command prints the list of process IDs to which it sent signals. The list of file systems that are being unmounted is also printed.

NOTES

The information displayed in Column 3 only appears if the file system was mounted as read-only.

NAME

pcnfsd – pc-nfs authentication and print spooling daemon

SYNOPSIS

/etc/pcnfsd [–d] [–s spooldir **]**

DESCRIPTION

The *pcnfsd* daemon processes authentication and print spool requests from MS-DOS clients running Sun Microsystem's PC-NFS. Requests and their responses are forwarded through the RPC/XDR(3N) package. Upon receipt of an authentication request, *pcnfsd* consults the server's password file and verifies that the password sent from the MS-DOS client matches that on the local machine. An acceptance or rejection message is then sent back to the client. The account's uid/gid pair is also returned if authentication succeeds. If the account in question does not exist, then authentication fails.

Print spooling consists of two services: spooling initialization and start print. When the server receives an initialization request, a path name for the print spool directory is assembled from *spooldir* and the client machine's name. This path is returned to the client. The client then NFS mounts this directory. When a spool file is ready to print, the start print request is sent to the server. The server then sends the file to the print spooling subsystem.

If the host handling *pcnfsd* service crashes, RPC timeout messages are returned to the user when the above requests are generated.

The following options are available:

–d Turn on debuging mode. Status messages are returned to the console terminal.

–s *spooldir*
The *pcnfsd* daemon uses *spooldir* instead of **/usr/spool/lp/pcnfsd** as the base spooling directory. This option is only available when *pcnfsd* is run as a daemon.

SEE ALSO

lp(1), passwd(1) in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual.*
getpwent(3C) in the *INTERACTIVE SDS Guide and Programmer's Reference Manual.*
lpr(1) in the *INTERACTIVE TCP/IP Guide.*

NAME

portmap – DARPA port to RPC program number mapper

SYNOPSIS

/etc/portmap

DESCRIPTION

The *portmap* server converts RPC program numbers into DARPA protocol port numbers. It must be running in order to make RPC calls.

When an RPC server is started, it will tell *portmap* what port number it is listening to and what RPC program numbers it is prepared to serve. When a client wishes to make an RPC call to a given program number, it will first contact *portmap* on the server machine to determine the port number where RPC packets should be sent.

SEE ALSO

rpcinfo(1M).

BUGS

If *portmap* crashes, all servers must be restarted.

NAME

rexd – RPC-based remote execution server

SYNOPSIS

/etc/rexd

DESCRIPTION

The *rexd* daemon is the Sun RPC server for remote program execution. For noninteractive programs, standard file descriptors are connected directly to TCP connections. Interactive programs involve pseudo-terminals, similar to the login sessions provided by *rlogin*(1). This daemon may use the NFS to mount file systems specified in the remote execution request.

FILES

/dev/ttyp*n*	pseudo-terminals used for interactive mode
/etc/passwd	authorized users
/tmp/rexd.log	if it exists, logs errors and events

SEE ALSO

mount(1M), on(1), rex(3), exports(4).

DIAGNOSTICS

Diagnostic messages are normally printed on the console and returned to the requester.

BUGS

Should be better access control.

NAME

rpcinfo – report RPC information

SYNOPSIS

rpcinfo **–p** [host]
rpcinfo **–u** host program [version]
rpcinfo **–t** host program [version]
rpcinfo **–b** program version

DESCRIPTION

The *rpcinfo* command makes an RPC call to an RPC server and reports what it finds.

The following options are available:

–p Probe the portmapper on *host* and print a list of all registered RPC programs. If *host* is not specified, it defaults to the value returned by *hostname*(1).

–u Make an RPC call to procedure 0 of *program* on the specified *host* using UDP and report whether a response was received.

–t Make an RPC call to procedure 0 of *program* on the specified *host* using TCP and report whether a response was received.

–b Make an RCP broadcast to procedure 0 of the specified *program* and *version* using UDP and report all hosts that respond.

The *program* argument can be either a name or a number.

If a version is specified, *rpcinfo* attempts to call that version of the specified *program*. Otherwise, *rpcinfo* attempts to find all the registered version numbers for the specified *program* by calling version 0 (which is presumed not to exist; if it does exist, *rpcinfo* attempts to obtain this information by calling an extremely high version number instead) and attempts to call each registered version. Note that the version number is required for the **–b** option.

EXAMPLES

To show all of the RCP services registered on the local machine, use:

example$ rpcinfo –p

To show all of the RCP services registered on the machine named **klaxon**, use:

example$ rpcinfo –p klaxon

To show all machines on the local network that are running the Network Information Service (NIS) (formerly known as the Yellow Pages) use:

example$ rpcinfo –b ypserv 'version' | uniq

where 'version' is the current NIS version obtained from the results of the **–p** switch above.

SEE ALSO

portmap(1M), rpc(4).
"INTERACTIVE NFS Protocol Specifications and User's Guide."

BUGS

In releases prior to SunOS 3.0, the Network File System did not register itself with the portmapper; *rpcinfo* cannot be used to make RPC calls to the NFS server on hosts running such releases.

NAME

rwall – write to all users over a network

SYNOPSIS

rwall hostname ...
rwall **–n** netgroup ...
rwall **–h** host **–n** netgroup

DESCRIPTION

rwall reads a message from standard input until end-of-file. It then sends this message, preceded by the line:

Broadcast Message ...

to all users logged in on the specified host machines. With the **–n** option, it sends to the specified network groups, which are defined in *netgroup*(4).

A machine can only receive such a message if it is running *rwalld*(1M), which is normally started up by the daemon *inetd*(1M).

SEE ALSO

rwalld(1M), netgroup(4).
wall(1) in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual.*
inetd(1M) in the *INTERACTIVE TCP/IP Guide*.

BUGS

The timeout is fairly short in order to be able to send to a large group of machines (some of which may be down) in a reasonable amount of time. Thus, the message may not get through to a heavily loaded machine.

NAME

rwalld – network rwall server

SYNOPSIS

/etc/rwalld

DESCRIPTION

rwalld is a server that handles *rwall*(1M) and *shutdown* (4.3BSD) requests. The *rwalld* daemon is normally invoked by *inetd*(1M).

SEE ALSO

rwall(1M).
inetd(1M), services(5) in the *INTERACTIVE TCP/IP Guide*.

NAME

showmount – show all remote mounts

SYNOPSIS

/etc/showmount [**–a**] [**–d**] [**–e**] [host]

DESCRIPTION

The *showmount* command lists all the clients that have remotely mounted a file system from *host*. This information is maintained by the *mountd*(1M) server on *host* and is saved across crashes in the file **/etc/rmtab**. The default value for *host* is the node name returned by *uname*(1).

The following options are available:

–d List directories that have been remotely mounted by clients.

–a Print all remote mounts in the format

hostname:directory

where *hostname* is the name of the client and *directory* is the **root** of the file system that has been mounted.

–e Print the list of exported file systems.

SEE ALSO

mountd(1M), exports(4), rmtab(4).
uname(1) in the *INTERACTIVE UNIX System User's/System Administrator's Reference Manual*.

BUGS

If a client crashes, its entry is not removed from the list until it reboots and actually *umount*s the file system (see *mount*(1M)).

NAME

statd – network status monitor

SYNOPSIS

/etc/statd [**–d** debuglevel]

DESCRIPTION

The *statd* daemon is an intermediate version of the status monitor. It interacts with *lockd*(1M) to provide the crash and recovery functions for the locking services on NFS.

The *statd* daemon preserves crash/recovery state in the **/etc/sm** directory. The *record* file records the host name of all currently monitored systems, the *recover* file records the host name of all systems that have as yet not been notified of *statd*'s failure, and the *state* file records the *statd*'s current version number.

The following option is available:

–d *debuglevel*

The *statd* daemon has extensive reporting internal reporting capabilities. A level of 2 reports significant events. A level of 4 reports internal state and all status monitor requests.

FILES

/etc/sm/record
/etc/sm/recover
/etc/sm/state

SEE ALSO

lockd(1M), statmon(4).

BUGS

The crash of a site is only detected upon its recovery.